SOMMARIO APPUNTI CORSO PYTHON

LEZIONE 1: Introduzione a Python ed installazione IDE (VSC – Virtual Studio Code) **1**

LEZIONE 2: Sintassi base di Python: commenti, indentazione, pulizia terminale..... **9**

LEZIONE 3: Le variabili. Sintassi variabili (camel case, pascal case, snake case). Assegnazione multipla delle variabili **11**

LEZIONE 4: Tipi di dati **13**

LEZIONE 5: Casting dei dati **16**

LEZIONE 6: Lavorare con le stringhe (metodi vari)..... **17**

LEZIONE 7: Valori Booleani (Boolean) **23**

LEZIONE 8: Operazioni aritmetiche..... **26**

LEZIONE 9: Ciclo if, elif, else. Operatori di comparazione. Short hand dell'if (scorciatoie) **29**

LEZIONE 10: Ciclo while. Break, continue, else. Interrompere un ciclo infinito (ctrl+c) **34**

LEZIONE 11: Ciclo for. Break, continue, else. For concatenati..... **37**

LEZIONE 12: Introduzione teorica sulle collezioni di dati: List, Tuple, Set, Dict (Dictionary). Differenze fra i quattro **41**

LEZIONE 13: List. Metodi vari **42**

LEZIONE 14: Tuple. Metodi vari..... **50**

LEZIONE 15: Set. Metodi vari **56**

LEZIONE 16: Dictionary (Dict). Metodi vari **61**

LEZIONE 17: Le funzioni. Definire i parametri. Differenza fra parametri ed argomenti. Arbitrary Arguments. Keyword Arguments. Arbitrary Keyword Arguments. Il return..... **71**

LEZIONE 18: Introduzione alla programmazione ad oggetti: classi ed oggetti. Il costruttore. Il costruttore self. Modificare i parametri di un oggetto. Eliminare proprietà di un oggetto. Eliminare oggetto **77**

LEZIONE 19: Ereditarietà. Funzione super. Metodi ed Overriding **82**

LEZIONE 20: Lo scope delle variabili. Scope locale e globale. Keyword global **87**

LEZIONE 21: I moduli. Importare modulo intero o una parte di modulo **90**

LEZIONE 22: Data e Ora con datetime. Esempi formattazione data **95**

LEZIONE 23: Modulo Math visto nel dettaglio. Matematica avanzata. Esempi di moduli per analisi dati..... **97**

LEZIONE 24: JSON per lo scambio dati. Leggere json in Python. Passare da Python a json. Ordinare le json..... **99**

LEZIONE 25: Installare nuovi moduli con pip. Disinstallazioni moduli aggiuntivi. Verifica pacchetti installati .**105**

LEZIONE 26: Try, Except, Finally. Errori multipli o multiple eccezioni. Raise/throw exception **109**

LEZIONE 27: Input degli utenti. Esercizio pratico con soluzione **112**

LEZIONE 28: Formattazione stringa avanzata. String format parameters. Inserimento valore multipli. Indici nominali **118**

LEZIONE 29: Lavorare con i file. Lavorare con i 4 parametri r, a, w, x. Leggere una parte di file. Leggere prima riga. Leggere sequenze di righe. Scrivere in un file. Sovrascrivere un file. Eliminare un file **120**

LEZIONE 30: Sito di riferimento per installare molti moduli importanti in Python, tipo NumPy per gli array .**128**

APPUNTI CORSO PYTHON

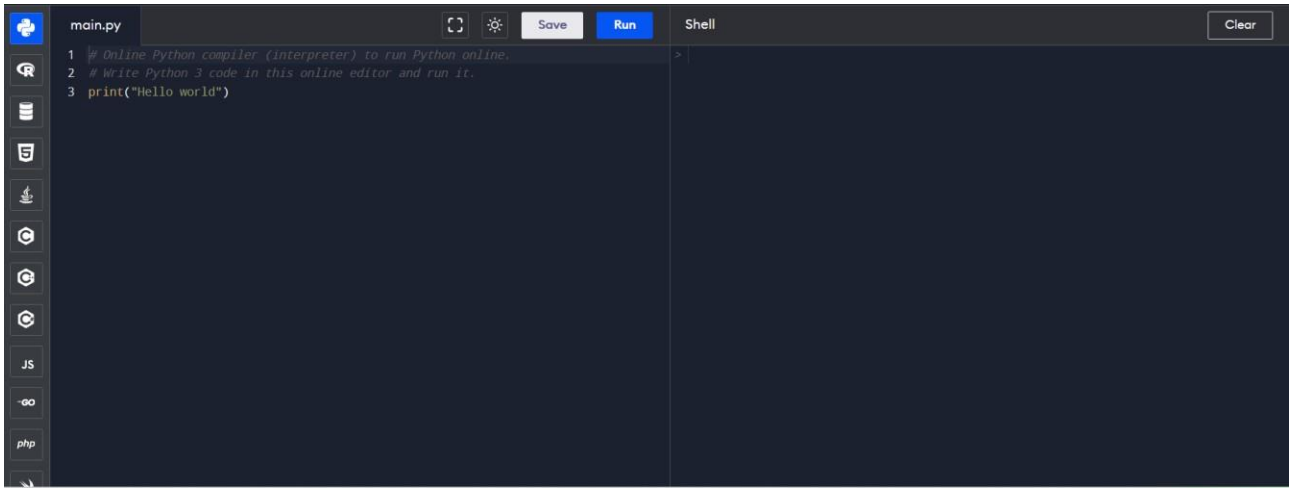
LEZIONE 1

INTRODUZIONE A PYTHON ED INSTALLAZIONE IDE:

Python è un linguaggio di alto livello. La sua sintassi, ovvero il modo in cui si scrive, è praticamente identica all'inglese scritto. Questo fa sì che rispetto ad altri linguaggi di programmazione come Java e C++, ricordare le varie istruzioni di Python sia estremamente semplice e rapido.

Per usare Python si può sfruttare il compilatore online presente nel link sotto:

https://www.programiz.com/python-programming/online-compiler/

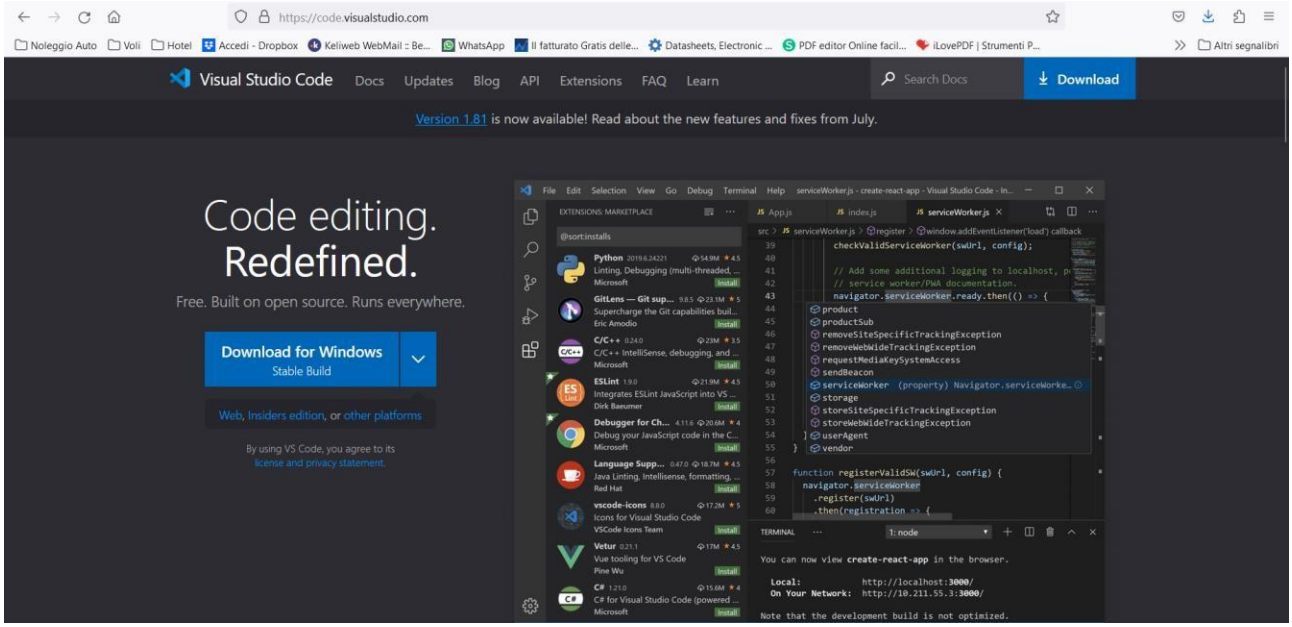


Oppure, come altamente consigliato, installiamo un compilatore sul nostro PC. Per installare Python andare su **https://www.python.org/** e cliccare su **download** e scaricare la versione per il vostro PC

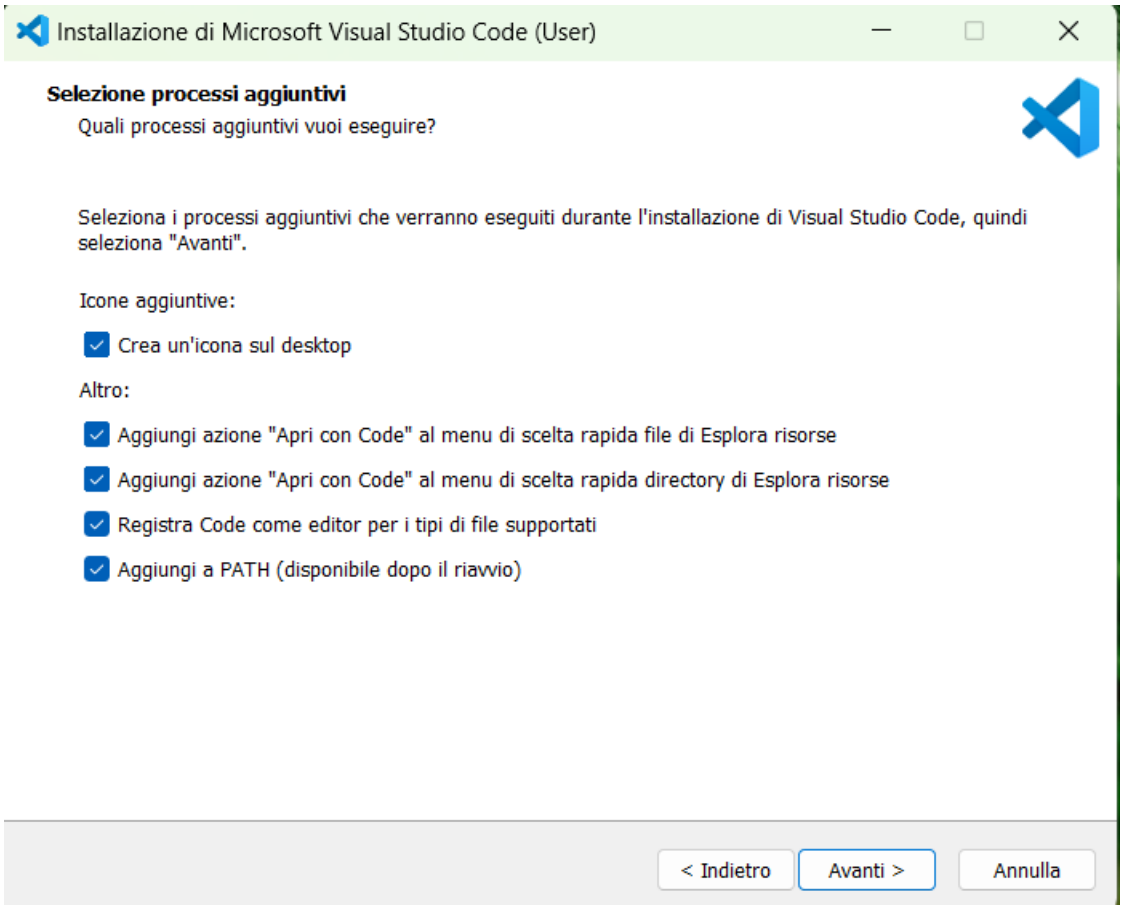
Lanciamo il nostro file .exe da installare e fare attenzione alla prima schermata a spuntare anche la casella del PATH, come sotto:

Per l'IDE, ovvero l'ambiente di sviluppo, utilizzeremo VSC (Visual Studio Code) disponibile a questo link

https://code.visualstudio.com/

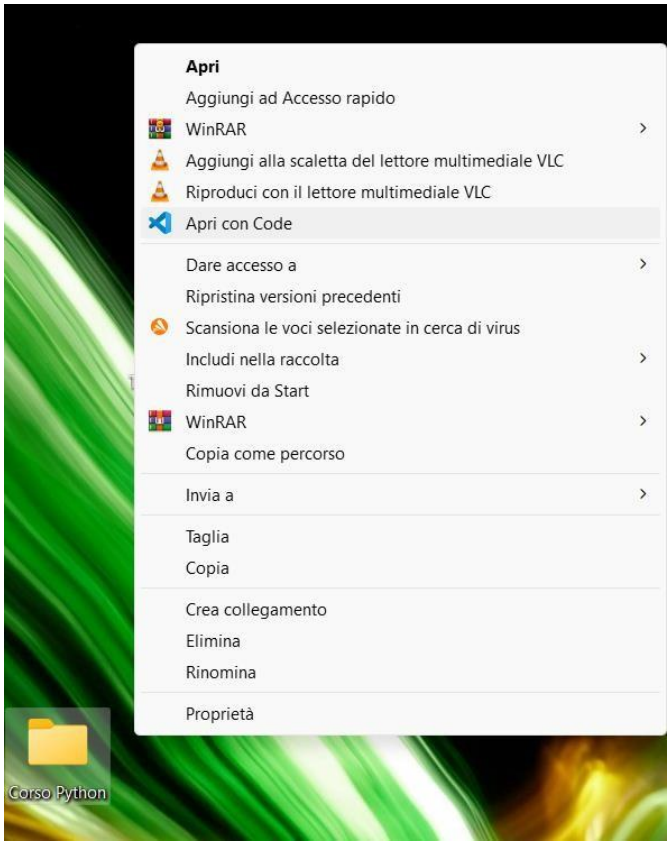


Durante l'installazione andare sempre avanti facendo attenzione a spuntare le caselle "Apri con code" come nella schermata sotto

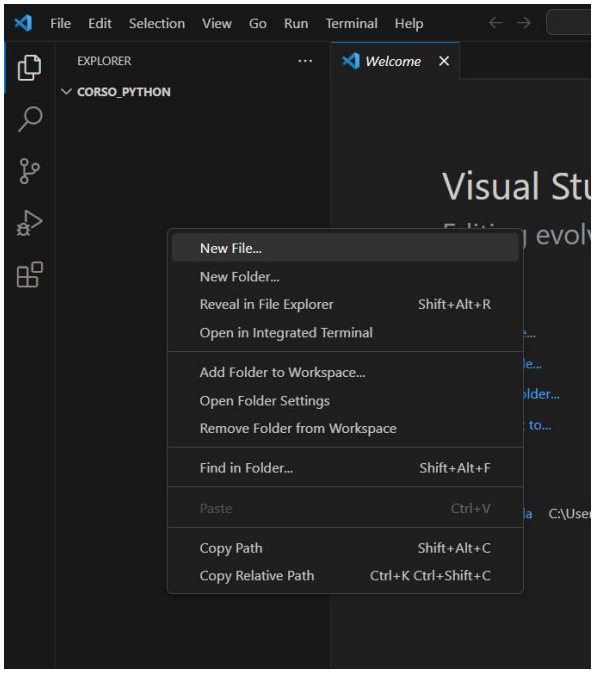


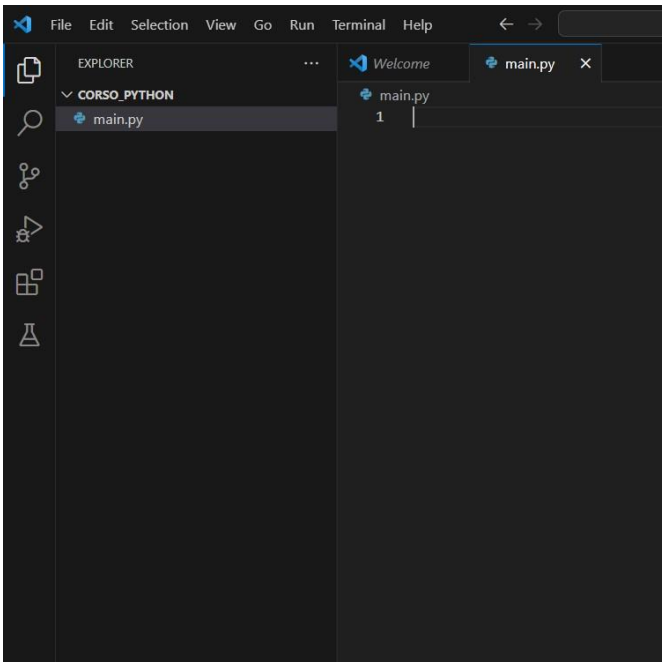
Finita l'installazione installiamo l'estensione di Python, dato che con VSC è possibile avere l'IDE di tantissimi linguaggi di programmazione. Clicchiamo sul tasto , cerchiamo Python ed installiamo l'estensione.

Ora, chiudiamo tutto, creiamo una nuova cartella sul Desktop ed apriamola con Code

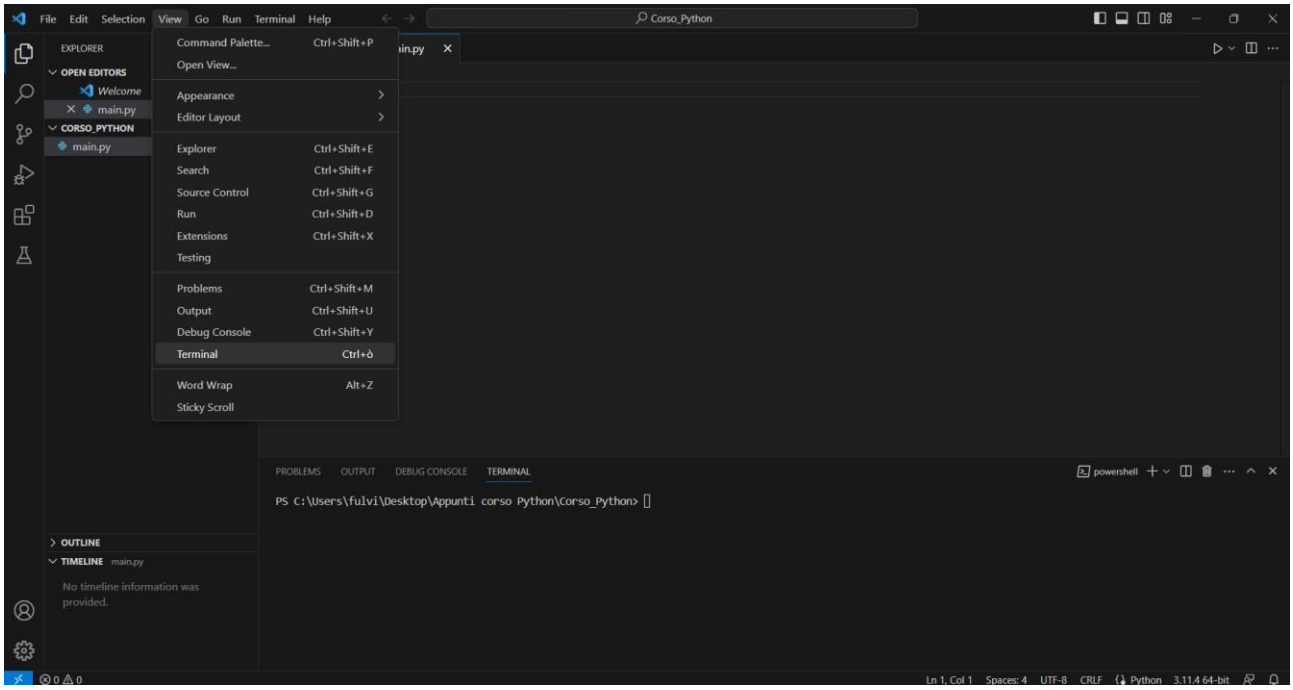


Cliccando col tasto destro creiamo un nuovo file e lo chiamiamo main.py

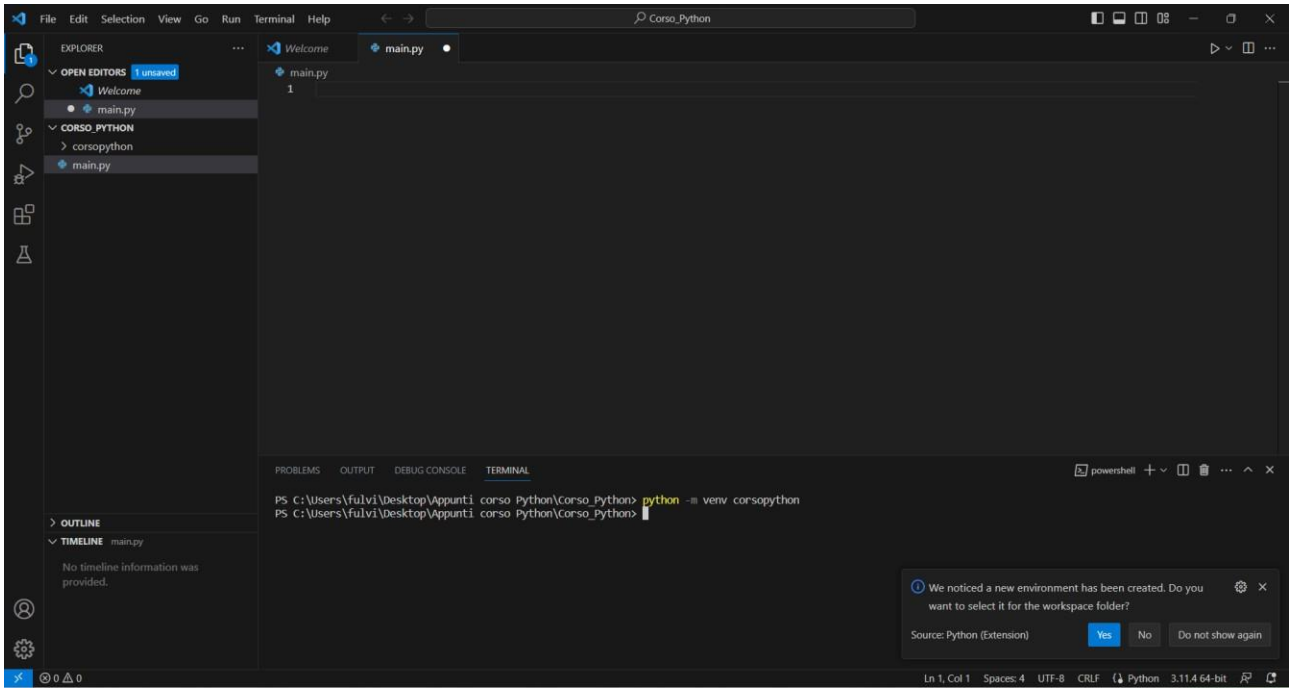




Dobbiamo ora creare un ambiente virtuale locale in base al lavoro che dobbiamo fare. Questo significa che andiamo ad installare particolari librerie o altro solo per i progetti interessati e non in maniera globale per tutti i progetti che abbiamo. In questo modo se dovessimo creare accidentalmente danni o altri tipi di problemi ed abbiamo necessità di cancellare alcune librerie, queste si cancelleranno solo sul progetto in questione senza intaccare gli altri progetti in essere. Visualizziamo il terminale da VIEW – TERMINAL

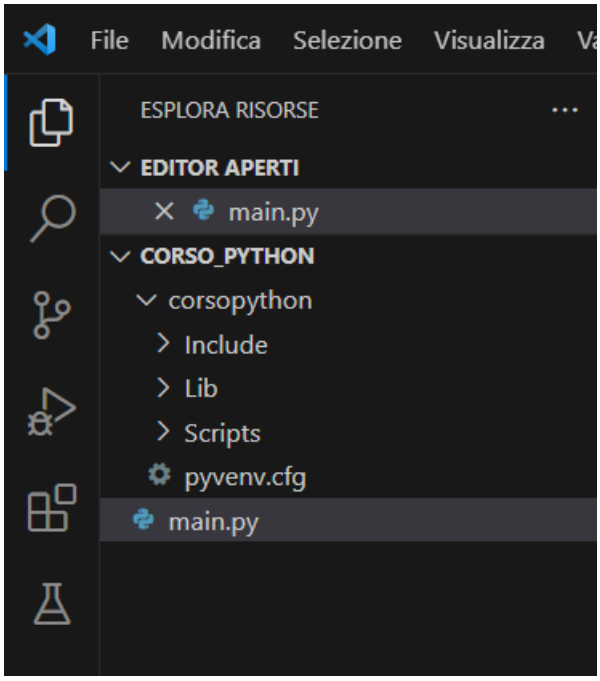


In terminal scriviamo **python -m venv corsopython** e premiamo invio

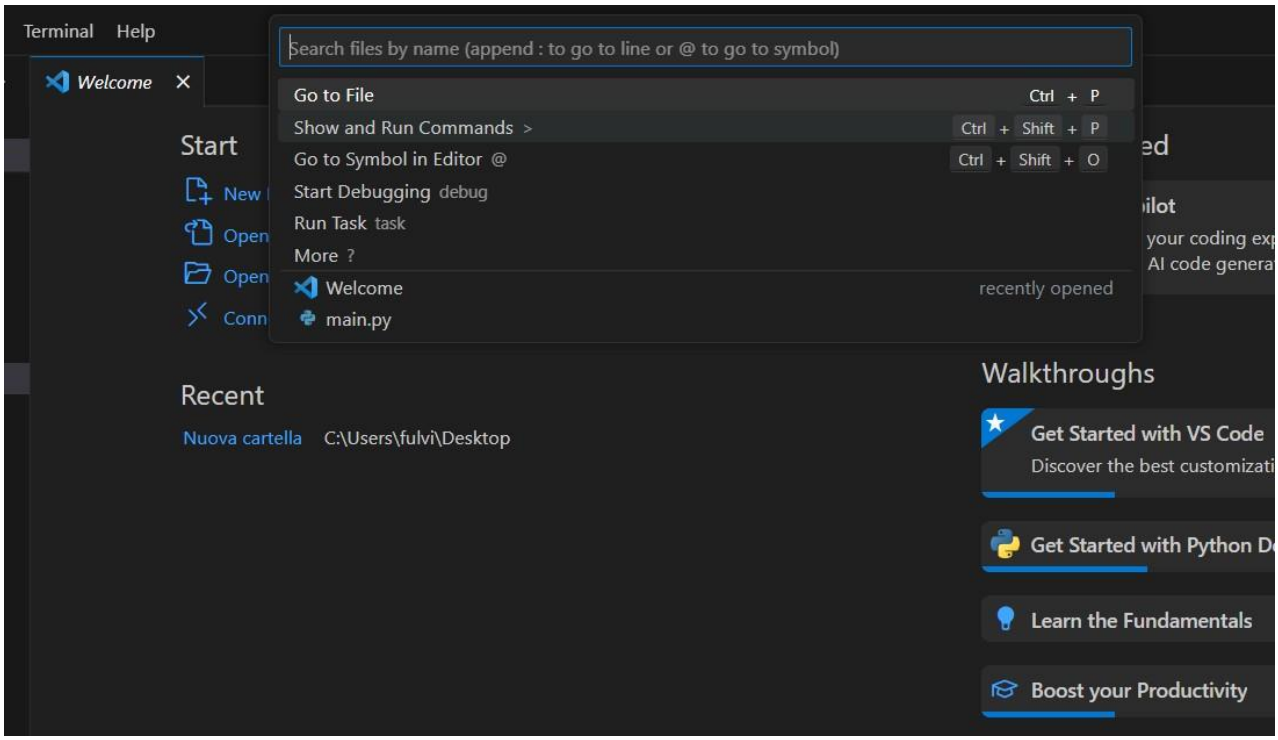


Ci chiederà se vogliamo gestirlo automaticamente. Rispondiamo NO per poterlo gestire a mano.

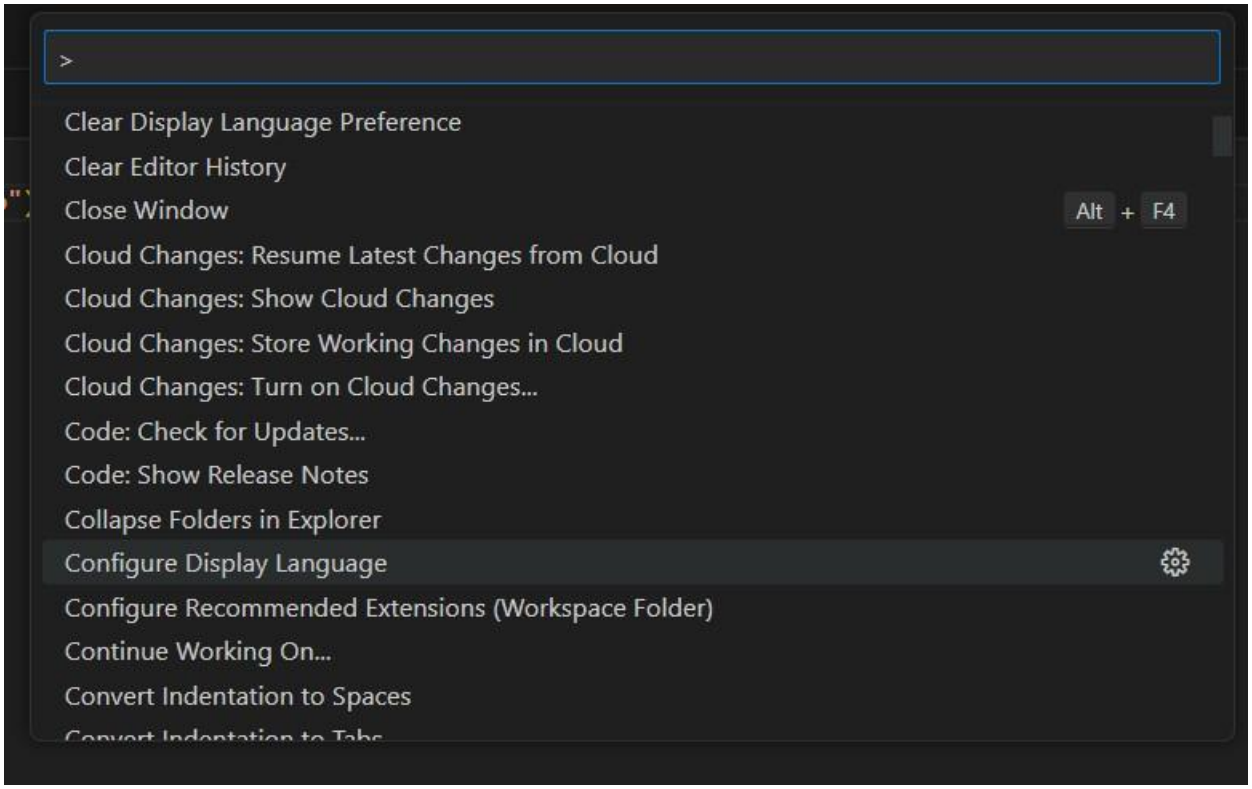
Vediamo sulla sinistra apparire le nostre librerie



Ora, prima di testare se la procedura è andata tutta ok, cambiamo lingua del programma (se non l'abbiamo già fatto) ed impostiamo Italiano. Per far cio basta cliccare nella barra in alto e selezionare "**SHOW AND RUN COMMANDS**" oppure digitare **CTRL + SHIFT + P**



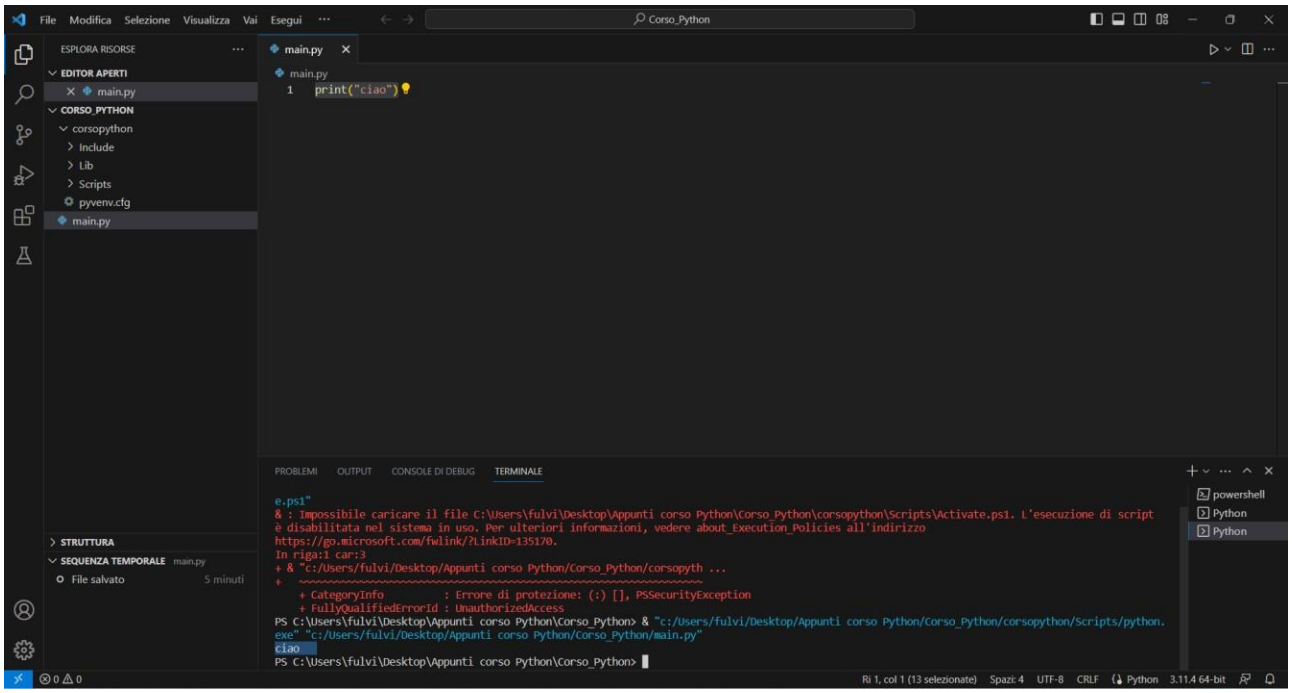
Fatto ciò, cerchiamo “**CONFIGURE DISPLAY LANGUAGE**” e selezioniamo l’Italiano



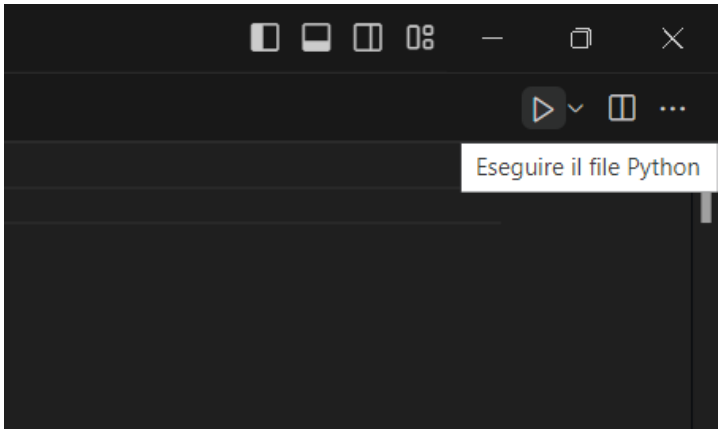
Scriviamo la prima riga di comando per testare il funzionamento e scriviamo

```
print("ciao")
```

per eseguire il programma clicchiamo col tasto destro e facciamo “**esegui file python nel terminale**” ed otterremo a schermo la nostra scritta *ciao.*



In alternativa, per eseguire il file basta cliccare sul tasto play in altro a destra, ottenendo il medesimo risultato



LEZIONE 2

SINTASSI BASE DI PYTHON:

= serve per commentare. Da tenere presente che non esistono commenti multi-riga in stile Java ma ad ogni riga commentata va aggiunto il

```
print("ciao") #commento
#commento
```

INDENTAZIONE (TAB) = Il rientro a capo e lo spazio (tab o indentazione) per python è l'essenza del programma. Mentre per alcuni linguaggi l'andare a capo e lasciare uno o più TAB di spazio è soprattutto per una visione e comprensione migliore del programma da parte del programmatore, in python incide sulla compilazione del codice stesso. Per cui se la tabulazione non è usata in maniera corretta il programma non funzionerà. Vediamo per esempio 2 istruzioni che possono sembrare simili ma che sono completamente diversi:

```
if 1<5:
    print("ciao")
```

```
if 1<5:
print("ciao")
```

Nel primo caso verrà stampato *ciao* perché l'istruzione fa parte dell'if da eseguire; nel secondo caso avremo un errore perché il *ciao* è un'istruzione a parte, fuori dall'if, per cui dentro l'if non c'è nessuna istruzione da eseguire nonostante abbiamo una condizione ed il programma darà un errore.

Per fare un esempio più pratico scriviamo.

```
if 1<5:
    print("ciao")
print("messaggio")
```

dato che 1<5 verrà stampata la condizione dentro l'if ed in oltre verrà stampato, a prescindere, il print fuori dall'if essendo una istruzione a parte

```
ciao
messaggio
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python>
```

Se invece abbiamo

```
if 10<5:
    print("ciao")
print("messaggio")
```

l'if non è verificato non verrà stampato il print che è una condizione dell'if mentre verrà comunque stampata la parola *messaggio* perché è una istruzione a parte

```
messaggio
```

Facciamo un esempio con identazione errata:

```
if 10<5:
print("ciao")
```

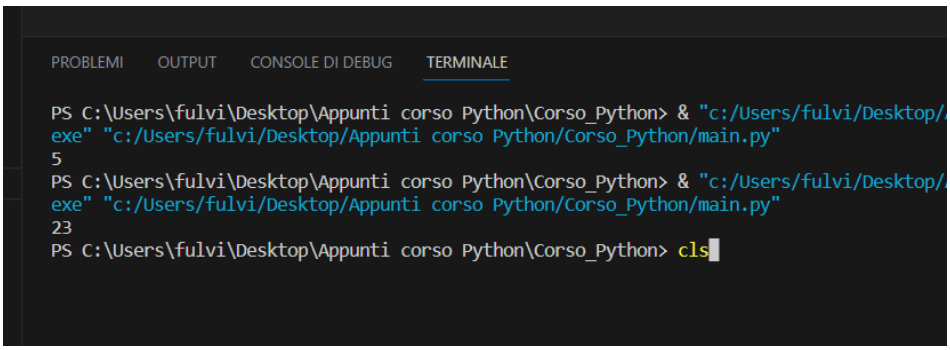
```
IndentationError: expected an indented block after 'if' statement on line 1
```

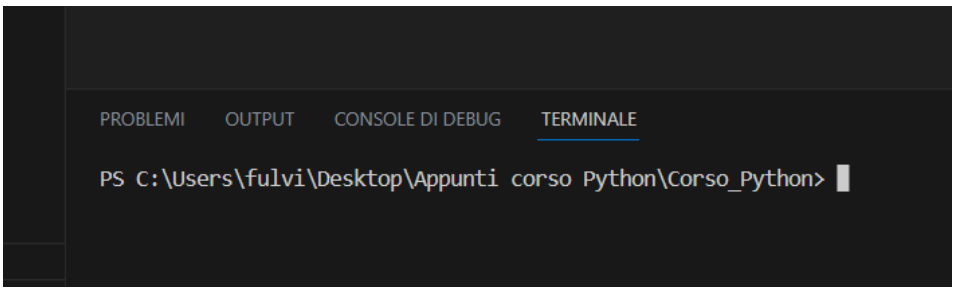
Da tenere presente, infine, che non ci sono **;** alla fine della sintassi come accade, ad esempio con Java. L'istruzione termina semplicemente dopo l'ultimo carattere scritto

```
print("Istruzione terminata. Non c'è bisogno di ;")
```

```
Istruzione terminata. Non c'è bisogno di ;
```

Se vogliamo **ripulire il terminale** che manda a video le nostre istruzioni basta scrivere nei comandi **cls** e premere invio





LEZIONE 3

LE VARIABILI:

Una variabile è un contenitore di uno o più valori (oggetti). È una scatola a cui faremo riferimento per prendere il valore. Python, a differenza degli altri linguaggi di programmazione, non ha bisogno che venga dichiarato che tipo di variabile è, ad esempio booleana, intera ecc… (che vedremo avanti) ma basta dichiarare nome e valore della variabile insieme ed il gioco è fatto (**dichiarazione ed assegnazione assieme**).

```
x = 5 #dichiarazione corretta. Var dichiarata e assegnata
y #dichiarazione errata. Var dichiarata ma non assegnata
```

Passiamo a vedere una parte fondamentale delle variabili, cioè la **nomenclatura** (ovvero regole utilizzate per dichiarare una variabile). Ad esempio, se abbiamo una variabile che rappresenta il peso di un cane ha più senso chiamarla pesoCane piuttosto che x, questo per permettere al programmatore di ricordare meglio ciò che ha scritto e di rendere comprensibile il programma a chi lo legge. Da tenere presente che python è case sensitive (sensibile a maiuscole e minuscole). Vediamo un esempio di una variabile che all'apparenza è sempre la stessa ma in realtà è diversa a seconda di come è dichiarata:

```
pesopersona
peso_persona
PesoPersona
PESOPERSONA
pesoPersona
#sono tutte variabili diverse a seconda della nomenclatura
```

Non è possibile dichiarare variabili che iniziano con numeri o che contengano trattini e spazi

```
1pesoPersona
peso persona
peso-persona
#sono tutte variabili dichiarate illegalmente. Non vanno bene
```

Per quanto riguarda i case possiamo avere il camel case, il pascal case e lo snake case.

Il **camel case** si ha quando una variabile inizia per minuscolo e se è formata da più parole la parola succeddiva avrà l'iniziale maiuscola.

```
pesoPersonaBrava
```

Il **pascal case** è uguale al camel case ma la prima parole inizia anche per maiuscola.

```
PesoPersonaBrava
```

Lo snake case consiste nel dividere le parole dal trattino basso (underscore):

```
peso_persone_brave
```

Vediamo ora come fare un'**assegnazione multipla delle variabili**, scrivere cioè tutto in un'unica riga invece che su righe differenti

```
x, y, z = 1, 2, 3
# equivale a scrivere
# x = 1
# y = 2
# z = 3
print(x)
print(y)
print(z)
```

```
1
2
3
```

Potremmo avere anche lo stesso valore per più variabili. In questo caso ci basta scrivere:

```
x = y = z = 1
print(x)
print(y)
print(z)
```

```
1
1
1
```

L'utilizzo di una variabile è molto semplice (come visto sopra). Facciamo un esempio:

```
x = y = 4
somma = x + y
print(somma)
```

```
8
```

Le variabili si possono usare più volte. Ad esempio:

```
x = 2
y = 3
somma = x + y + x + x
print(somma)
```

```
9
```

Nelle lezioni successive vedremo come utilizzare le variabili con altre funzioni e operazioni

LEZIONE 4

TIPI DI DATI:

Come già accennato nelle lezioni precedenti, in Python possiamo inizializzare ed assegnare un valore ad un dato senza però sapere che tipo di dato è, diversamente da altri linguaggi dove è obbligatorio dichiarare il tipo di dato. Ora se noi scriviamo X = 5 e mandiamo a schermo, otteniamo il risultato 5. È corretto avere 5 a schermo ma non sappiamo, però, ancora che tipo di dato è 5. La funzione per sapere il tipo di dato è **type(x)**

```
x = 5
print(type(x))
```

```
<class 'int'>
```

Questa volta a schermo non otteniamo il valore del dato ma bensì il tipo. Il tipo di dato viene assegnato automaticamente nel momento in cui il programmatore va a dare un valore alla variabile.

Vediamo i principali tipi di dati:

str: Stringa, cioè un testo

int: Numero intero

float: Numero con la virgola

bool: Booleano, ovvero vero o falso, True o False. Fare attenzione a scrivere True e False con la lettera maiuscola. Si usa soprattutto quando dobbiamo verificare delle condizioni

list: Lista; a livello sintattico uguale agli array ma non sono array. Gli array verranno trattati più in là con la libreria di NumPy. Le liste sono collezioni di dati, insieme di valori. I dati vengono inseriti in parentesi quadra

tuple: Sono collezioni di dati diversi o uguali, contrassegnati da un indice e non modificabili. Differiscono dalle liste per le parentesi utilizzate. Qua i dati vengono inseriti in parentesi tonda

range: È un range assegnato in automatico. Ad esempio se scriviamo X = range(6) verranno assegnati automaticamente dei valori che vanno da 0 a 5 (6 indici). In pratica crea una lista di 6 valori automaticamente.

dict: Dictionary. Si creano come liste e tuple ma utilizzano le parentesi graffe. Utilizzano un rapporto chiave-valore, ad ogni chiave è associato un valore (approfondiremo nelle lezioni future)

set: E' una sequenza di elementi unici non ordinati, separati da una virgola e racchiusi all'interno di due parentesi graffe { }. In altre parole, un set è una lista che: non contiene duplicati e non mantiene l'ordine con cui inserisci gli elementi.

Facciamo un esempio pratico:

```
x = "ciao"
print(type(x))
print(x)
# str

x = 5
print(type(x))
print(x)
#int

x = 3.2
print(type(x))
print(x)
#float

x = True
print(type(x))
print(x)
#bool

x = ["Roma", "Napoli", "Milano"]
print(type(x))
print(x)
#list

x = ("Roma", "Napoli", "Milano")
print(type(x))
print(x)
#tuple

x = range(6)
print(type(x))
print(x)
#range

x = {"nome": "Luca", "età": 25}
print(type(x))
print(x)
#dict

x = {"Roma", "Napoli", "Milano"}
print(type(x))
print(x)
#set
```

A video avremo:

```
<class 'str'>
ciao
<class 'int'>
5
<class 'float'>
3.2
<class 'bool'>
True
<class 'list'>
['Roma', 'Napoli', 'Milano']
<class 'tuple'>
('Roma', 'Napoli', 'Milano')
<class 'range'>
range(0, 6)
<class 'dict'>
{'nome': 'Luca', 'età': 25}
<class 'set'>
{'Napoli', 'Milano', 'Roma'}
```

In sintesi:

```
# str: x = "ciao"
# int: x = 5
# float: x = 3.2
# bool: x = True

# list: x = ["Roma", "Napoli", "Milano"]
# tuple: x = ("Roma", "Napoli", "Milano")
# range: x = range(6)
# dict: x = {"nome": "Luca", "età": 25}
# set: x = {"Roma", "Napoli", "Milano"}
```

Dato che liste, tuple, range, dictionary e set sembrano uguali, li distingueremo in base alle loro proprietà. Ognuno di questi tipi verrà approfondito nelle lezioni seguenti.

LEZIONE 5

CASTING:

Vuol dire prendere un valore di una variabile e convertirlo in un altro valore, ad esempio prendere un numero e convertirlo in una stringa. È utile per concatenare 2 tipi di dati differenti. Vediamo un esempio:

```
x = "5"
y = 5
print(x + y)
#otterremo un errore perchè non si può concatenare una stringa ad un intero
```

In alcuni linguaggi si concatenano in automatico le variabili x e y mentre in Python ci sono alcune operazioni da fare. Vediamo, ad esempio come concatenare una stringa ed un intero.

```
x = "Ciao "
y = str(5)
print(x + y)
```

```
Ciao 5
```

Adesso siamo riusciti a concatenarli perché ad y abbiamo assegnato un intero del valore 5 convertito in stringa. Proviamo a fare il contrario, ovvero convertire una stringa in intero:

```
x = int("2")
y = 2
print(x + y)
```

```
4
```

Altro esempio con float()

```
x = float(5)
print(x)
```

```
5.0
```

Questa volta abbiamo convertito l'intero 5 in float 5.0

LEZIONE 6

LAVORARE CON LE STRINGHE:

Per creare una stringa si possono usare o le virgolette o i singoli apici. Per convenzione è sconsigliato usare in un programma entrambe le forme, meglio usare sempre la stessa formattazione.

```
x = "ciao"
y = 'ciao'
```

Si può mandare a schermo o direttamente la stringa o la variabile che la contiene:

```
x = "ciao"
print(x)
# mandiamo a schermo la variabile

print("ciao")
# mandiamo a schermo direttamente la stringa
```

Da notare che usando doppi apici o singoli apici possiamo scrivere una **stringa singola riga**. Se vogliamo scrivere una stringa su più righe (**stringa multi riga**) dobbiamo usare tripli doppi apici oppure tripli singoli apici:

```
x = """ciao
vado
a
capo"""
print(x)
```

```
ciao
vado
a
capo
```

Vediamo ora come trattare le stringhe come degli array (prendere carattere, length, loop); ovvero una stringa è una collezione di caratteri per cui ogni stringa è la successione dei singoli caratteri. Se vogliamo **stampare il singolo carattere** basta mandare a schermo la variabile da stampare con l'indice del carattere che vogliamo stampare. Ricordiamo che gli indici sono gli indirizzi dei singoli caratteri e che partono sempre da zero.

```
x = "ciao"
print(x[0])
# manda a schermo l'indice 0 della stringa
```

```
c
```

Anche lo spazio ovviamente è considerato come singolo carattere:

```
x = "ciao a tutti"
print(x[7])

# indici: 012345678
#         ciao a te
```

```
t
```

Per vedere la **lunghezza di una stringa**, quindi da quanti caratteri è formata, basta scrivere len(x):

```
x = "ciao a tutti"
print(len(x))
```

```
12
```

Da notare che la lunghezza è il numero effettivo di caratteri e parte da 1. Quello che parte da 0 è l'indice. Non confondersi con questi due termini. In pratica l'indice è la lunghezza meno 1.

```
x = "ciao a tutti"
print(len(x))
print(x[11])
```

```
12
i
```

Possiamo fare il **loop della stringa**, ovvero mandare a schermo singolo carattere per singolo carattere. Si capirà meglio più avanti quando vedremo il ciclo for:

```
for x in "ciao":
    print(x)
```

```
c
i
a
o
```

Vediamo come prendere **parte di una stringa**:

```
x = "ciao a tutti"

print(x[0:3])
print()
# stampa dall'indice 0 al 3 (non compreso)

print(x[3:])
print()
# stampa dall'indice 3 fino alla fine

print(x[3:8])
# stampa dall'indice 3 all'indice 8 (non compreso)
```

```
cia

o a tutti

o a t
```

Possiamo anche prendere **parte di stringhe con indici negativi**:

```
x = "ciao a tutti"

print(x[-5])
print()
# stampa dalla fine e parte da -1, -2, -3, ec...

print(x[-5:-2])
# stampa da -5 a -2 (non compreso)
```

```
t

tut
```

Andiamo ora a modificare una stringa utilizzando dei metodi:

upper() ci restituisce tutto maiuscolo

```
x = "ciao a tutti"
print(x.upper())
```

```
CIAO A TUTTI
```

lower() ci restituisce tutto minuscolo:

```
x = "CIAO"
print(x.lower())
```

```
ciao
```

strip() serve per togliere gli spazi bianchi che abbiamo davanti e dietro all'interno di una stringa. Utile quando si devono scrivere delle password:

```
x = "        ciao a tutti "
print(x.strip())
```

```
ciao a tutti
```

replace() sostituisce un carattere nella stringa:

```
x = "ciao a tutti"
print(x.replace("i", "www"))
```

```
cwwwao a tuttwww
```

split() serve a dividere una stringa in una lista di stringhe contenenti tante stringhe quante sono le parole nella stringa iniziale

```
x = "ciao a tutti"
print(x.split())
```

```
['ciao', 'a', 'tutti']
```

Concatenare le stringhe:

```
x = "ciao a "
y = "tutti"
print(x + y)
```

```
ciao a tutti
```

Ora introduciamo il concetto di **format()** che ci aiuta a combinare stringhe e numeri. Diversamente da altri linguaggi, in Python non possiamo concatenare una stringa con un numero a meno che all'interno della stringa non mettiamo le parentesi graffe ad indicare i punti che ci interessano, e quando andiamo a stampare a schermo all'interno del metodo format() scriviamo il numero da concatenare alla stringa. Vediamo un esempio:

```
x = "ciao sono Paolo e sono nato il {}"
print(x.format(30))
```

```
ciao sono Paolo e sono nato il 30
```

La stessa cosa si può fare anche come variabile:

```
x = "ciao sono Paolo e sono nato il {}"
y = 23
print(x.format(y))
```

```
ciao sono Paolo e sono nato il 23
```

Vediamo un esempio di **concatenazione in più punti tramite format()**:

```
x = "ciao sono Paolo e sono nato il {} e peso {}"
print(x.format(15, 65))
```

```
ciao sono Paolo e sono nato il 15 e peso 65
```

Possiamo anche variare gli indici delle parentesi graffe e mettere i numeri in base alle nostre esigenze. Senza nulla nelle parentesi graffe, nell'esempio sopra, abbiamo di default indici 0 e 1. Se vogliamo stamparli invertiti basta fare:

```
x = "ciao sono Paolo e sono nato il {1} e peso {0}"
print(x.format(15, 65))
```

```
ciao sono Paolo e sono nato il 65 e peso 15
```

Vediamo infine l'**escape dei caratteri**. Per esempio, se vogliamo scrivere all'interno di una stringa qualcosa tra virgolette otterremo un errore se la parte tra virgolette non è inserita tra \ (backslash). Vediamo prima un esempio con errore:

```
x = "ciao sono Paolo e sono "figo""
print(x)
```

```
SyntaxError: invalid syntax
```

Scriviamo ora la sintassi corretta:

```
x = "ciao sono Paolo e sono \"figo\""
print(x)
```

```
ciao sono Paolo e sono "figo"
```

Se usiamo i singoli apici ci capiterà il problema con gli apostrofi. Vediamo un esempio con errore:

```
x = 'sono alla ricerca dell'amore'
print(x)
```

```
SyntaxError: unterminated string literal (detected at line 1)
```

Riscriviamolo ora in forma corretta:

```
x = 'sono alla ricerca dell\'amore'
print(x)
```

```
sono alla ricerca dell'amore
```

Questo è solo un esempio dell'escape dei caratteri. In Python possiamo trovare molti altri escape.

LEZIONE 7

BOOLEAN:

I valori booleani sono, come già visto, il vero o falso. Si utilizzano molto negli If, nei cicli ecc…

```
x = True
y = False
```

Da notare che True o False devono essere scritte con la lettera maiuscola. Proviamo a fare un esempio:

```
print (5 < 10)
```

```
True
```

Vediamo che a schermo ci darà True perché effettivamente la condizione è verificata. Facciamo un esempio concreto con un if (che verrà approfondito nelle lezioni successive). Non faremo altro che porre una condizione, ad esempio 5 < 10, se è verificata (True) stamperemo qualcosa altrimenti (False) ne stamperemo un'altra.

```
if 5 < 10:
    print("minore di 10")
else:
    print("maggiore di 10")
```

```
minore di 10
```

Ora vediamo come **valutare i valori delle variabili**:

```
print(bool(0))
print()
print(bool(1))
```

```
False

True
```

Quindi il primo valore restituirà False mentre il secondo restituirà True. La regola generale è che verrà restituito sempre True tranne nei casi:

False:

bool(false) # variabile falsa

bool(None) # variabile None, non trovata

bool(0) # numero 0

bool(“”) # stringa vuota

bool(()) # tuple vuote

bool([]) # list vuota

bool({}) # set vuota

Facciamo un ulteriore esempio:

```
pane = 0
if pane:
    print("pane in dispensa, non serve comprarlo")
else:
    print("uscire a comprare il pane")

# se il pane è presente stampa l'if, altrimenti stampa l'else
# dato che pane è uguale a 0, quindi false, stampa l'else
```

```
uscire a comprare il pane
```

Altro esempio con le liste:

```
lista_cose_da_comprare = []
if lista_cose_da_comprare:
    print("andare al supermercato")
else:
    print("non serve andare al supermercato")

# se la lista di cose da comprare è vuota stampa " non serve andare al
supermercato"
# altrimenti stampa "andare al supermercato"
```

```
non serve andare al supermercato
```

Nel caso di sopra la lista di cose da comprare è vuota per cui non serve andare al supermercato, nel caso di sotto invece è piena e serve andare al supermercato:

```
lista_cose_da_comprare = ["pane", "patate", "limoni"]
if lista_cose_da_comprare:
    print("andare al supermercato")
else:
    print("non serve andare al supermercato")
```

```
andare al supermercato
```

LEZIONE 8

OPERAZIONI ARITMETICHE:

Vediamo gli **operatori aritmetici di base**:

+ = somma

- = sottrazione

***** = moltiplicazione

/ = divisione

% = modulo, ovvero il resto della divisione

****** = potenza

// = floor division, ovvero risultato della divisione arrotondato per difetto

```
x = 9
y  =  5
print(x + y)
print(x - y)
print(x / y)
print(x * y)
print(x % y)
print(x ** y)
print(x // y)
```

```
14
4
1.8
45
4
59049
1
```

Se vogliamo inserire un messaggio e dare il risultato dell'operazione all'interno dello stesso print ricordiamo che dobbiamo usare la concatenazione, ovvero concatenare una stringa (il messaggio) ed un intero (il risultato dell'operazione)

```
x = 9
y = 5
print("la somma è = " + str(x + y))
```

```
la somma è = 14
```

Ovviamente se abbiamo espressioni, varrà la stessa regola degli operatori matematici (prima moltiplicazioni e divisioni, poi somma e sottrazioni)

```
print(2 + 2 / 2)
# ovviamente verrà eseguita prima la divisione e poi la somma
```

```
3.0
```

Andiamo a vedere ora gli **operatori di assegnamento**:

```
x = 5
x = 5 + 2
print(x)
```

```
7
```

Con gli operatori di assegnamento c'è un modo più semplice di riscrivere la formula sopra, ovvero:4

```
x = 5
x += 2
print(x)
# x += 2 equivale proprio a x = x + 2
```

```
7
```

Gli operatori di assegnamento valgono per la somma così come per gli altri operatori (+=, -=, *=, ecc…). Ci sono altri operatori di assegnamento che verranno visti più avanti.

Vediamo alcuni **metod**i utili già presenti in Python:

min = restituisce il minimo di una serie di valori

```
x = min(5, 10, 23)
print(x)
```

```
5
```

max = restituisce il massimo di una serie di valori

```
x = max(5, 10, 23)
print(x)
```

```
23
```

abs = absolute, restituisce il valore assoluto di un numero

```
x = abs(-5)
print(x)
```

```
5
```

pow = potenza

```
x = pow(5,2)
print(x)
```

```
25
```

Per fare formule matematiche più complesse basta importare il modulo math, che però vedremo più avanti.

LEZIONE 9

CONDIZIONI CON IF, ELIF, ELSE:

Serve per dare una condizione. O si esegue questa istruzione o quest'altra in base alla condizione che imponiamo al programma. Cominciamo con una condizione semplice (**if semplice**):

```
if 5 < 10:
    print("5 minore di 10")
    # esegue questa istruzione solo nel
    # caso la condizione 5 < 10 sia vera
```

```
5 minore di 10
```

In questo caso la condizione è verificata e manda in stampa l'istruzione, se non era verificata, come sotto, non mandava in stampa nulla. Come già anticipato nelle lezioni precedenti stare attenti alla tabulazione e mettere i : dopo l'if

```
if 11 < 10:
    print("5 minore di 10")
    # esegue questa istruzione solo nel
    # caso la condizione 5 < 10 sia vera
```

Se vogliamo mandare a schermo qualcosa che non è nell'if basta uscire dalla tabulazione:

```
x = 5
if x < 10:
    print("x minore di 10")
print("non sono nell'if")
```

```
x minore di 10
non sono nell'if
```

ed ancora

```
x = 12
if x < 10:
    print("x minore di 10")
print("non sono nell'if")
```

```
non sono nell'if
```

In entrambi i casi viene stampato "non sono nell'if" perché, appunto, è una istruzione a se al di fuori dell'if; nel primo caso viene stampato anche "x minore di 10" perché la condizione interna all'if è verificata.

Andiamo a vedere ora gli **operatori di comparazione**:

== : uguale

!= : diverso

< : minore

> : maggiore

>= : maggiore uguale

<= : minore uguale

```
x = 10
if x == 10:
    print("condizione verificata")
# condizione uguale

x = 20
if x != 10:
    print("condizione verificata")
# condizione diverso

x = 20
if x > 10:
    print("condizione verificata")
# condizine maggiore

x = 5
if x < 10:
    print("condizione verificata")
# condizine minore

x = 10
if x >= 10:
    print("condizione verificata")
# condizine maggiore uguale

x = 5
if x <= 5:
    print("condizione verificata")
# condizine minore uguale
```

```
condizione verificata
condizione verificata
condizione verificata
condizione verificata
condizione verificata
condizione verificata
```

Introduciamo ora l'**else**, ovvero se non si verifica una condizione allora esegui quest'altra istruzione.

```
x = 20
if x < 10:
    print("condizione verificata")
else:
    print("condizione non verificata")
```

```
condizione non verificata
```

Da tenere sempre presente la regola della spaziatura con il tab, che è simile all'if, ed i : per eseguire l'istruzione. Se dobbiamo mettere più condizioni allora usiamo **elif**, che sta a significare else if. La regola del tab e dei : resta sempre valida.

```
x = 20
if x < 10:
    print("x minore di 10")
elif x>30:
    print("x maggiore di 30")
else:
    print("x compreso tra 10 e 30")
```

```
x compreso tra 10 e 30
```

Da tenere presente che quando si valutano più condizioni bisogna mettere un if, un else ed infiniti elif.

Vediamo velocemente una sintassi molto abbreviata che non vi è in altri linguaggi di programmazione. Ad esempio, se vogliamo vedere se un numero è compreso tra 10 e 20 possiamo semplicemente scrivere:

```
x = 15
if 10 < x < 20:
    print("x compreso tra 10 e 20")
```

```
x compreso tra 10 e 20
```

L'esempio di sopra si può risolvere anche utilizzando l'and. Introduciamo quindi gli **operatori logici and, or e not**. Per verificare se x è compreso tra una condizione ed un'altra condizione si usa and.

```
x = 15
if x > 10 and x < 20:
    print("x compreso tra 10 e 20")
```

```
x compreso tra 10 e 20
```

Perché si mandi a stampa il messaggio, l'and presuppone che siano verificate entrambe le condizioni (con and devono essere verificate per forza sia x >10 che x < 20).

Vediamo un ulteriore esempio:

```
x = 15
y = 7
if x > 10 and y > 10:
    print("verificata")
else:
    print("NON verificata")
```

```
NON verificata
```

Andiamo ora ad utilizzare l'or. In questo caso per entrare nell'if basta che una sola condizione sia verificata e non tutte come accadeva con l'and.

```
x = 15
y = 7
if x > 10 or y > 10:
    print("verificata")
else:
    print("NON verificata")
```

```
verificata
```

Infine, abbiamo il not. Il not va a negare la condizione.

```
x = 15
if not(x < 10):
    print("verificata")
else:
    print("NON verificata")
```

```
verificata
```

La condizione x < 10 non è verificata in quanto x è 15, per cui non dovremmo entrare nell'if ma stampare l'else; dato che però abbiamo usato il not abbiamo negato la condizione, per cui si entra nell'if e non nell'else. In parole povere x < 10 è false, ma noi stiamo facendo una negazione di una negazione che significa true.

X < 10 not(false) => true e di conseguenza è una condizione verificata e si entra nell'if.

Passiamo ora alla versione **Short Hand dell'if**. Per short hand si intende una scorciatoia. Un esempio è il seguente:

```
x = 15
if x > 10: print("verificata")
```

```
verificata
```

Come notiamo dopo i : dell'if abbiamo il print. Questo si può fare solo e soltanto se abbiamo una sola istruzione. Non è possibile mettere 2 istruzioni dopo i due punti. Altro short hand che si può usare è con if else (in realtà si usa poco):

```
x = 15
print("è maggiore di 10") if x > 10 else print("è minore di 10")
```

```
è maggiore di 10
```

Passiamo agli **if innestati o annidati** ovvero if dentro gli if. Facciamo un esempio. Vogliamo sapere se un numero è maggiore di 10 ed è pari. Potremmo usare un and ma ai fini del nostro scopo useremo 2 if.

```
x = 9
if x > 10:
    if x % 2 == 0:
        print("numero pari e maggiore di 10")
    else:
        print("numero maggiore di 10 ma dispari")
else:
    print("numero minore di 10")
```

```
numero minore di 10
```

cambiando il valore della x:

```
x = 12
if x > 10:
    if x % 2 == 0:
        print("numero pari e maggiore di 10")
    else:
        print("numero maggiore di 10 ma dispari")
else:
    print("numero minore di 10")
```

```
numero pari e maggiore di 10
```

LEZIONE 10

CICLO WHILE:

I loop o cicli non sono altro che un ciclo infinito di operazioni tutti uguali finché la condizione non è verificata. Se, ad esempio, abbiamo una lista con n valori, per stampare tutti i valori non ha senso utilizzare tanti print quanti sono i valori della lista perché diventerebbe insostenibile ed improduttivo, come nell'esempio sotto:

```
x = ["Milano", "Roma", "Napoli"]
print(x[0])
print(x[1])
print(x[2])

# se abbiamo una lista e vogliamo stampare tutti i valori
# non ha senso scrivere tanti print quanti sono i valori
# ma bisogna utilizzare i cicli
```

```
Milano
Roma
Napoli
```

Abbiamo bisogno quindi di un iteratore, ovvero una variabile che tiene traccia del punto in cui siamo arrivati. Vediamo l'esempio sotto, tenendo presente che la sintassi per il while è uguale a quella dell'if

```
i = 0
while i < 6:
    print(i)
    i += 1
```

```
0
1
2
3
4
5
```

Vediamo cosa fa il ciclo while. Per prima cosa vediamo che i=0, per cui è rispettata la condizione i<6 e viene stampata i, quindi stampato 0. Successivamente con l'istruzione i += 1 cambiamo valore di i. Infatti il nuovo valore è 0 +1 per cui i=1 sempre minore di 6 (la nostra condizione) e stampiamo 1. Successivamente i diventa 2 e viene stampato 2 e così via fin quando i=6 e non viene più rispettata la condizione i<6. In quel momento si ferma il ciclo ed esegue l'istruzione al di fuori del while (se ci sono altre istruzioni).

Andiamo ora a vedere altre 3 parole importanti per il while, ovvero **break, continue, else**. Cominciamo con il break. Il break serve ad interrompere il ciclo anche se il while non è finito. Vediamo l'esempio sotto:

```
i = 0
while i < 6:
    print(i)
    if i ==3:
        break
    i += 1
print("ciclo terminato")
```

```
0
1
2
3
ciclo terminato
```

In questo caso, come già anticipato, il break interrompe il ciclo quando i è uguale a 3 e quindi si esce fuori dal while e si continua con l'esecuzione del resto del programma, in questo caso print("ciclo terminato").

Se per errore si lancia un loop infinito nel terminale dei comandi e si vuole **interrompere il ciclo in terminal** basta cliccare **CTRL+C**

Vediamo ora invece il continue con un esempio non ottimale ma che ne mostra chiaramente il funzionamento:

```
i = 0
while i < 6:
   i += 1
   if i ==3:
        continue
   print(i)

print("fine ciclo")
```

```
1
2
4
5
6
fine ciclo
```

In questo caso anche se i=0 non viene stampato perché rispetta la condizione i<6 ma la prima istruzione è incrementare i di 1. Quindi i=1 e si entra nell'if dove la condizione è i==3, che chiaramente non è rispettata per cui si esce dall'if e si esegue la stampa di i. Poi si incrementa i che diventa 2 e si stampa 2. Si incrementa ancora i che diventa 3 ma questa volta viene rispettata anche la condizione dell'if, ovvero i==3; si esegue dunque l'istruzione continue che significa vai avanti senza tener conto di i==3, quindi non stamparla. Si incrementa ancora i che diventa 4 e si continua fino al risultato sopra.

Passiamo infine all'else. Questo funziona come per l'if, una volta finito il ciclo di while vai avanti ed esegui l'istruzione contenuta nell'else. Vediamo un esempio:

```
i = 0
while i < 6:
   print(i)
   i += 1
else:
    print("ciclo finito")
```

```
0
1
2
3
4
5
ciclo finito
```

Riassumendo, la sintassi basica del ciclo while è quella sottostante:

```
i = 0
while i < 6:

   i += 1
```

tutto il resto viene inserito in base alle esigenze del programmatore. L'importante è ricordarsi sempre di incrementare il contatore altrimenti andiamo avanti fino all'infinito.

LEZIONE 11

CICLO FOR:

Il for varia dal while in quanto il for serve per iterare una sequenza. Mentre con il while avevamo una condizione, con il for si itera una sequenza che è già stata creata. Facciamo un **esempio con una lista**:

```
lista_citta = ["Milano", "Torino", "Napoli"]

for i in lista_citta:
    print(i)
```

```
Milano
Torino
Napoli
```

Non abbiamo fatto altro che creare la lista lista_citta contenente 3 stringhe. Per stampare tutta la lista creiamo un ciclo for con un iteratore chiamato i (si può chiamare in qualunque modo) che scandisce tutti gli elementi presenti in lista_citta e li manda a schermo con print(i). Se aggiungiamo una città nella lista non dobbiamo fare nulla per trovarla a schermo, infatti il for itera sempre tutti gli elementi interni alla lista salvo diverse indicazioni dell'utente.

```
lista_citta = ["Milano", "Torino", "Napoli", "Venezia", "Roma"]

for i in lista_citta:
    print(i)
```

```
Milano
Torino
Napoli
Venezia
Roma
```

Vediamo un altro **esempio di for con una stringa**.

```
stringa = "anguria"

for lettera in stringa:
    print(lettera)
```

```
a
n
g
u
r
i
a
```

Facciamo un esempio di **for con il range**:

```
for i in range(6):
    print(i)

#stampiamo tutti i valori da 0 a 5 (6 valori appunto)
```

```
0
1
2
3
4
5
```

Come per il while, anche per il for si può utilizzare **break, continue, else**. Vediamo un esempio di break

```
for i in range(6):
    if i == 3:
        break
    print(i)

# interrompe il ciclo quando i = 3
```

```
0
1
2
```

Ora vediamo un esempio con continue:

```
for i in range(6):
    if i == 3:
        continue
    print(i)

# salta il numero 3
```

```
0
1
2
4
5
```

Vediamo un esempio con else:

```
for i in range(6):
    print(i)
else:
    print("ciclo finito")
```

```
0
1
2
3
4
5
ciclo finito
```

Da notare che quando c'è il break il ciclo termina ed esce sia dal for che dall'else:

```
for i in range(6):
    if i == 3:
        break
    print(i)
else:
    print("ciclo finito")
```

```
0
1
2
```

Infatti dato che c'è il break questa volta oltre a fermarsi prima del 3 non stampa neanche quello che c'è nell'else

Vediamo come fare **for concatenati**. Ad esempio, se vogliamo stampare una tabella 2x3 (2 righe, 3 colonne) abbiamo bisogno di 2 for, uno che scandisca le righe e l'altro le colonne.

```
for riga in range(2):
    for colonna in range(3):
        print("riga:" + str(riga) + " colonna:" + str(colonna))
```

```
riga:0 colonna:0
riga:0 colonna:1
riga:0 colonna:2
riga:1 colonna:0
riga:1 colonna:1
riga:1 colonna:2
```

In pratica stiamo rappresentando la situazione sottostante:

	col0	**col1**	**col2**
riga0:	0	1	2
riga1:	0	1	2

LEZIONE 12

INTRODUZIONE TEORICA SULLE COLLEZIONI DI DATI: LIST, TUPLE, SET e DICTIONARY:

Come già visto le collezioni di dati sono le liste, le tuple, il set ed il dictionary. Nelle prossime lezioni verranno spiegati uno per uno ma per capire bene la differenza fra i 4 è utile metterli tutti a confronto. Intanto ricordiamo che una variabile può avere un solo valore ma se vogliamo più valori dobbiamo utilizzare una delle 4 collezioni di dati sopra indicati. Abbiamo già visto che le collezioni di dati sono racchiuse nelle parentesi ed il tipo di parentesi dipende dal tipo di collezione che andiamo a creare e quindi dalle proprietà che li caratterizzano. Una **collezione può essere**:

ordinata: la collezione ha un ordine ben definito e l'aggiunta di nuovi elementi non incide sulla collezione stessa

indicizzata: si può accedere ad un elemento tramite indice (index). Qualsiasi cosa ordinata è anche indicizzata

modificabile: si possono aggiungere, rimuovere e cambiare elementi una volta creata la collezione

immutabile: non si possono aggiungere, rimuovere e cambiare elementi una volta creata la collezione. Modificabile ed immutabile vanno di pari passo anche se sono una l'opposto dell'altra

permette duplicati: possono esserci più elementi con lo stesso valore

Fatta questa premessa andiamo a vedere la **differenza fra le 4 collezioni di dati**:

list: collezioni ordinate (e quindi indicizzate) e modificabili. Permettono duplicati

tuple: collezioni ordinate (e quindi indicizzate) ma non modificabili. Permettono duplicati

set: collezioni non ordinate e perciò non indicizzate. Non permettono duplicati

dictionary: collezioni ordinate (dalla versione 3.7) e modificabili. Non permettono duplicati

Nelle prossime 4 lezioni verranno viste le 4 collezioni singolarmente

LEZIONE 13

LIST:

La lista è la versione più flessibile di tutte le collezioni di dati. Crearla è molto semplice ed è stata vista più volte.

```
x = ["Milano," "Napoli", "Roma"]
```

Si può creare una lista mista contenente ad esempio, stringhe, numeri e booleani

```
x = ["Milano," 100, True]
```

Useremo **type()** per vedere il tipo di dato:

```
x = [50, 100, 25]
print(type(x))
```

```
<class 'list'>
```

Per vedere la lunghezza basta usare **len()** e ci verrà restituito la quantità di elementi presenti nella lista:

```
x = [50, 100, 25]
print(len(x))
```

```
3
```

Si può creare una lista anche usando il costruttore **list()**, fare attenzione a come si usano le parentesi tonde:

```
y = list(("Milano", "Roma", "Napoli"))
```

Vediamo come **accedere agli elementi della lista** tramite gli indici:

```
x = [50, 100, 25]
print(x[1])
# accesso all'elemento con indice 1. Quantità 3 elementi con indici da 0 a 2
```

```
100
```

Per **prendere l'ultimo elemento** basta mandare a schermo l'indice -1:

```
x = [50, 100, 25]
print(x[-1])
```

```
25
```

Se vogliamo usare gli indici a ritroso basta prendere -2 per il penultimo, -3 per il terzultimo e così via.

Ora vediamo come **stampare range di lista**:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
print(x[2:])  # Napoli, Venezia
print(x[:3])  # Milano, Roma, Napoli
print(x[1:3]) # Roma, Napoli
```

```
['Napoli', 'Venezia']
['Milano', 'Roma', 'Napoli']
['Roma', 'Napoli']
```

Per **modificare gli elementi** basta prendere l'indice o il range di indice che ci interessa e modificare gli elementi:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x[1]= "Firenze"
print(x)
```

```
['Milano', 'Firenze', 'Napoli', 'Venezia']
```

Altro esempio:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x[1:3]= ["Firenze", "Lamezia"]
print(x)
```

```
['Milano', 'Firenze', 'Lamezia', 'Venezia']
```

Per **inserire nuovi elementi** si usa **append()**, **insert()**, **extend()**.

append() inserisce l'elemento in fondo alla lista:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x.append("Brescia")
print(x)
```

```
['Milano', 'Roma', 'Napoli', 'Venezia', 'Brescia']
```

insert() inserisce l'elemento secondo un indice fornito, quindi nella posizione desiderata:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x.insert(1, "Brescia")
print(x)
```

```
['Milano', 'Brescia', 'Roma', 'Napoli', 'Venezia']
```

extend() invece estende una lista con un'altra lista:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = ["Brescia", "Udine"]

x.extend(y)

print(x)
print(y)
```

```
['Milano', 'Roma', 'Napoli', 'Venezia', 'Brescia', 'Udine']
['Brescia', 'Udine']
```

Notiamo che y è rimasta uguale mentre x è cambiata in quanto è stata estesa con y.

Vediamo adesso come rimuovere degli elementi utilizzando un po' di metodi: **remove()**, **pop()**, **del()**, **clear()**.

remove() rimuove un elemento specificandolo. Da tenere presente che è case sensitive, per cui prestare attenzione a maiuscole e minuscole:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x.remove("Milano")

print(x)
```

```
['Roma', 'Napoli', 'Venezia']
```

pop() elimina l'ultimo elemento se non ha indice al suo interno, altrimenti elimina l'elemento corrispondente all'indice dentro le parentesi:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x.pop()

print(x)
```

```
['Milano', 'Roma', 'Napoli']
```

Altro esempio:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x.pop(2)

print(x)
```

```
['Milano', 'Roma', 'Venezia']
```

In alternativa si può usare del().

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
del x[0]

print(x)
```

```
['Roma', 'Napoli', 'Venezia']
```

Scrivendo solo **del x** rimuoviamo tutta la lista, quindi la lista non esisterà più.

Utilizzando clear() invece puliremo la lista. La lista esisterà ancora ma sarà vuota.

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x.clear()

print(x)
```

```
[]
```

Andiamo ora a ciclare gli elementi nelle liste.

Mandare in stampa tutti gli elementi con il for:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
for citta in x:
    print(citta)
```

```
Milano
Roma
Napoli
Venezia
```

Potremmo scrivere lo stesso codice creando un indice i che cicla un range della lunghezza della lista x e mandi a schermo tutti gli indici della lista

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
for i in range(len(x)):
    print(x[i])
```

```
Milano
Roma
Napoli
Venezia
```

Utilizziamo ora il while:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
i = 0
while i < len(x):
    print(x[i])
    i = i+1
```

```
Milano
Roma
Napoli
Venezia
```

Vediamo come **modificare l'ordine della lista**:

sort() ordina la lista alfabeticamente o numericamente:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
x.sort()
print(x)
```

```
['Milano', 'Napoli', 'Roma', 'Venezia']
```

Altro esempio:

```
x = [15, 8, 3, 2, 101, 47, 85]
x.sort()
print(x)
```

```
[2, 3, 8, 15, 47, 85, 101]
```

sort(reverse=True) ordina la lista alfabeticamente o numericamente al contrario:

```
x = [15, 8, 3, 2, 101, 47, 85]
x.sort(reverse=True)
print(x)
```

```
[101, 85, 47, 15, 8, 3, 2]
```

Per altri metodi simili consultare i manuali con i comandi di Python. Al momento abbiamo visto solo i più comuni.

Andiamo a vedere come **copiare una lista** con il comando **copy()**. Potremmo non usare il copy() e scrivere semplicemente che una lista è = ad un'altra

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = x

print("la lista x è: " + str(x))
print("la lista y è: " + str(y))
```

```
la lista x è: ['Milano', 'Roma', 'Napoli', 'Venezia']
la lista y è: ['Milano', 'Roma', 'Napoli', 'Venezia']
```

Il problema però è che se cambio un elemento di y si cambia anche lo stesso elemento di x, come vediamo sotto, questo perché in realtà y fa un riferimento di x e non una copia

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = x
y[0] = "Udine"

print("la lista x è: " + str(x))
print("la lista y è: " + str(y))
```

```
la lista x è: ['Udine', 'Roma', 'Napoli', 'Venezia']
la lista y è: ['Udine', 'Roma', 'Napoli', 'Venezia']
```

Usando il metodo copy() ovviamo a questo problema:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = x.copy()
y[0] = "Udine"

print("la lista x è: " + str(x))
print("la lista y è: " + str(y))
```

```
la lista x è: ['Milano', 'Roma', 'Napoli', 'Venezia']
la lista y è: ['Udine', 'Roma', 'Napoli', 'Venezia']
```

in alternativa x.copy() si può usare list(x)

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = list(x)
y[0] = "Udine"
print("la lista x è: " + str(x))
print("la lista y è: " + str(y))
```

```
la lista x è: ['Milano', 'Roma', 'Napoli', 'Venezia']
la lista y è: ['Udine', 'Roma', 'Napoli', 'Venezia']
```

Vediamo, infine, come **unire più liste**. Possiamo utilizzare il **+**:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = ["Bari", "Lamezia Terme"]

print(x + y)
```

```
['Milano', 'Roma', 'Napoli', 'Venezia', 'Bari', 'Lamezia Terme']
```

Possiamo utilizzare **append()** però in questo caso dobbiamo usare il for per ciclare tutti gli elementi:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = ["Bari", "Lamezia Terme"]

for citta in y:
    x.append(citta)

print(x)
```

```
['Milano', 'Roma', 'Napoli', 'Venezia', 'Bari', 'Lamezia Terme']
```

Un ultimo metodo per unire 2 liste è usare **extend()**:

```
x = ["Milano", "Roma", "Napoli", "Venezia"]
y = ["Bari", "Lamezia Terme"]

x.extend(y)

print(x)
```

```
['Milano', 'Roma', 'Napoli', 'Venezia', 'Bari', 'Lamezia Terme']
```

Altri metodi aggiuntivi si possono trovare nella documentazione online di Python.

LEZIONE 14

TUPLE:

Come già detto le tuple sono collezioni di dati ordinati, indicizzati, non modificabili e che permettono duplicati. In alcune cose sono uguali alle liste (tipo l'indicizzazione) per cui tali argomenti verranno solo trattati velocemente, mentre verranno approfonditi gli argomenti che differiscono dalle liste.

Creazione di una tupla (uguale alle liste ma con le parentesi tonde):

```
x = ("Milano", "Roma", "Napoli", "Venezia")
print(x)
```

```
('Milano', 'Roma', 'Napoli', 'Venezia')
```

Si possono avere sia un tipo di valore che valori mischiati:

```
x = ("Milano", "Roma", "Napoli", "Venezia", 100, False)
```

La cosa da sapere è che si ha una **tupla di un solo valore**, serve una virgola dopo il valore altrimenti non viene riconosciuta come tupla ma come tipo, ad esempio stringa o intero ecc…

```
x = ("Milano",)
print(type(x))
# questa è riconosciuta come tupla perchè c'è la virgola

y = ("Milano")
print(type(y))
# questa non è riconosciuta come tupla perchè non c'è la virgola
# è riconosciuta solo per il tipo, in questo caso stringa
```

```
<class 'tuple'>
<class 'str'>
```

Metodi Len() e Type() per definire lunghezza e tipo:

```
x = ("Milano", "Roma", "Napoli")
print(len(x))
print(type(x))
```

```
3
<class 'tuple'>
```

Costruttore tuple() per dichiarare in maniera alternativa un tupla (fare attenzione all'uso delle parentesi tonde):

```
x = tuple(("Milano", "Roma", "Napoli"))
print(x)
```

```
('Milano', 'Roma', 'Napoli')
```

Accedere agli elementi singoli o range, così come per le liste:

```
x = tuple(("Milano", "Roma", "Napoli"))

print(x[0])
print(x[0:2])
print(x[:1])
```

```
Milano
('Milano', 'Roma')
('Milano',)
```

Verificare se un elemento esiste:

```
x = tuple(("Milano", "Roma", "Napoli"))

if "Milano" in x:
    print("Elemento trovato")
```

```
Elemento trovato
```

Non è possibile modificare o rimuovere gli elementi ammenoché non si usa un escamotage. Andiamo intanto a modificare un elemento ed otterremo un errore:

```
x = tuple(("Milano", "Roma", "Napoli"))
x[0] = "Venezia"
print(x)
```

```
TypeError: 'tuple' object does not support item assignment
```

L'**escamotage per modificare la tupla** è prendere la tupla, trasformarla in lista, modificarla e poi ritrasformarla in tupla. Lo stesso procedimento vale per aggiungere e rimuovere un elemento:

```
x = ("Milano", "Roma", "Napoli")
y = list(x)
# trasformiamo la tupla in lista

y[0] = "Venezia"
# sostituiamo l'indice 0, ovvero Venezia con Milano

x = tuple(y)
# riconvertiamo la lista in tupla

print(x)
# stampiamo la tupla modificata
```

```
('Venezia', 'Roma', 'Napoli')
```

Per eliminare definitivamente una tupla basta usare **del()**:

```
x = ("Milano", "Roma", "Napoli")
del x
print(x)
# a schermo avremo un errore perchè la tupla x è stata eliminata
# quindi a schermo non otterremo nulla
```

```
NameError: name 'x' is not defined
```

Vediamo come **spacchettare una tupla**, ovvero singolo elemento per singolo elemento:

```
citta = ("Milano", "Roma", "Napoli")

(x, y, z) = citta

print(x)
print(y)
print(z)
```

```
Milano
Roma
Napoli
```

Nel caso abbiamo più elementi e prenderne solo una parte basta mettere un asterisco(*) davanti all'ultimo pezzo da spacchettare:

```
citta = ("Milano", "Roma", "Napoli", "Venezia")

(x, y, *z) = citta
# x e y sono singoli elementi mentre z diventa una lista

print(x)
print(y)
print(z)
```

```
Milano
Roma
['Napoli', 'Venezia']
```

Altro esempio:

```
citta = ("Milano", "Roma", "Napoli", "Venezia")

(x, *y, z) = citta
# x e z sono singoli elementi mentre y diventa una lista

print(x)
print(y)
print(z)
```

```
Milano
['Roma', 'Napoli']
Venezia
```

Vediamo ora come **ciclare gli elementi di una tupla**. È uguale alle liste. Iniziamo con il ciclo for:

```
citta = ("Milano", "Roma", "Napoli", "Venezia")

for i in citta:
    print(i)
```

```
Milano
Roma
Napoli
Venezia
```

Altro esempio utilizzando il range():

```
citta = ("Milano", "Roma", "Napoli", "Venezia")

for i in range(len(citta)):
    print(citta[i])
```

```
Milano
Roma
Napoli
Venezia
```

Vediamo ora il while:

```
citta = ("Milano", "Roma", "Napoli", "Venezia")

i = 0
while i < len(citta):
    print(citta[i])
    i += 1
```

```
Milano
Roma
Napoli
Venezia
```

Per **unire le tuple** basta usare il **+**:

```
x = ("Milano", "Roma", "Napoli", "Venezia")
y = ("Torino", "Bari")
print(x + y)
```

```
('Milano', 'Roma', 'Napoli', 'Venezia', 'Torino', 'Bari')
```

Oppure in alternativa potremmo creare un'altra lista che unisce le due:

```
x = ("Milano", "Roma", "Napoli", "Venezia")
y = ("Torino", "Bari")
z = x + y
print(z)
```

```
('Milano', 'Roma', 'Napoli', 'Venezia', 'Torino', 'Bari')
```

Per contare quante volte un elemento è presente in una tupla usiamo il metodo **count()**:

```
lista_citta = ("Milano", "Roma", "Napoli", "Venezia", "Milano")

x = lista_citta.count("Milano")

print(x)
```

```
2
```

Per vedere qual è l'indice di un valore che mettiamo nella tupla bisogna usare index():

```
lista_citta = ("Milano", "Roma", "Napoli", "Venezia", "Milano")

x = lista_citta.index("Milano")

print(x)
```

```
0
```

Da notare che, se un elemento è presente più volte verrà restituito solo l'indice del primo elemento trovato.

In definitiva possiamo affermare che le 2 grandi differenze tra liste e tuple è l'uso delle parentesi, quadre per liste e tonde per tuple, e che le tuple non sono modificabili mentre le liste si.

LEZIONE 15

SET:

Come già i set sono collezioni di dati non ordinati, non indicizzati, non modificabili e che non permettono duplicati. Possiamo **creare un set** sia con un unico tipo di dato sia con dati differenti. A differenza di tuple e liste, in questo caso vengono usate le parentesi graffe:

```
x = {"Milano", "Venezia", "Roma"}
y = {"Bari", 34, True}
```

Vediamo i soliti metodi **type(), len()**:

```
x = {"Milano", "Venezia", "Roma"}
print(type(x))
print(len(x))
```

```
<class 'set'>
3
```

Anche qui è presente il **costruttore set()** prestando sempre attenzione al numero di parentesi tonde utilizzate:

```
x= set(("Milano", "Roma", "Napoli"))
print(x)
```

```
{'Milano', 'Roma', 'Napoli'}
```

Non essendo ne ordinate ne indicizzate, per **accedere agli elementi** si può usare solo il **loop**; infatti, se mandiamo a schermo x ogni volta ci escono valori diversi nonostante la riga di istruzione sia sempre la stessa.

```
x = {"Milano", "Venezia", "Roma"}

print(x)
```

```
{'Roma', 'Milano', 'Venezia'}
```

```
{'Milano', 'Roma', 'Venezia'}
```

Se invece proviamo ad accede ad un elemento otterremo un errore:

```
x = {"Milano", "Venezia", "Roma"}
print(x[0])
```

```
TypeError: 'set' object is not subscriptable
```

Vediamo allora come fare il loop per accedere agli elementi:

```
x = {"Milano", "Venezia", "Roma"}

for i in x:
    print(i)
```

```
Roma
Milano
Venezia
```

Per **vedere se un elemento esiste**:

```
x = {"Milano", "Venezia", "Roma"}
print("Milano" in x)
```

```
True
```

Per **aggiungere gli elementi** si usa **add()** oppure **update()**. Vediamo add():

```
x = {"Milano", "Venezia", "Roma"}
x.add("Bari")

print(x)
```

```
{'Milano', 'Bari', 'Roma', 'Venezia'}
```

Vediamo update():

```
x = {"Milano", "Venezia", "Roma"}
y = {"Napoli", "Firenze"}
x.update(y)

print(x)
```

```
{'Milano', 'Venezia', 'Firenze', 'Roma', 'Napoli'}
```

Ora vediamo come **rimuovere gli elementi** con diversi metodi. Iniziamo da **remove()**:

```
x = {"Milano", "Venezia", "Roma"}
x.remove("Milano")

print(x)
```

```
{'Venezia', 'Roma'}
```

Vediamo **discard()**:

```
x = {"Milano", "Venezia", "Roma"}
x.discard("Milano")

print(x)
```

```
{'Venezia', 'Roma'}
```

All'apparenza remove() e discard() sembrano uguali mentre in realtà la differenza è che andiamo a rimuovere un elemento che non esiste, con remove avremo un errore mentre con discard non succederà nulla.

Si può rimuovere l'ultimo elemento con **pop()** però bisogna stare attenti che essendo il set non ordinato l'elemento che verrà tolto è casuale:

```
x = {"Milano", "Venezia", "Roma"}
x.pop()

print(x)
```

```
{'Roma', 'Milano'}
```

Abbiamo poi il metodo **clear()** che restituisce un **set vuoto**:

```
x = {"Milano", "Venezia", "Roma"}
x.clear()

print(x)
```

```
set()
```

Oppure possiamo **eliminare completamente il set** con **del**:

```
x = {"Milano", "Venezia", "Roma"}
del x

print(x)
```

```
NameError: name 'x' is not defined
```

Passiamo ad **unire gli elementi**. Cominciamo con il metodo **union()**. Union creerà un nuovo set:

```
x = {"Milano", "Venezia", "Roma"}
y ={"Napoli", "Udine"}
z = x.union(y)

print(z)
```

```
{'Udine', 'Napoli', 'Milano', 'Roma', 'Venezia'}
```

Vediamo invece **update()**, che a differenza di union non crea un nuovo set ma aggiunge solamente gli elementi:

```
x = {"Milano", "Venezia", "Roma"}
y ={"Napoli", "Udine"}
x.update(y)

print(x)
```

```
{'Napoli', 'Venezia', 'Milano', 'Roma', 'Udine'}
```

Sia update() che union() vanno ad escludere gli elementi duplicati, riportandoli una sola volta. Per **lavorare sugli elementi in comune di due set** serve il metodo **intersection_update()**. Con questo metodo ci verranno restituiti solo gli elementi comuni alle due liste e verrà aggiornato un set già presente:

```
x = {"Milano", "Venezia", "Roma"}
y ={"Napoli", "Milano"}
x.intersection_update(y)

print(x)
```

```
{'Milano'}
```

Allo stesso modo può essere usato **intersection()** con la differenza che questa volta verrà creato un nuovo set:

```
x = {"Milano", "Venezia", "Roma"}
y ={"Napoli", "Milano"}
z = x.intersection(y)

print(z)
```

```
{'Milano'}
```

Vediamo infine i metodi **symmetric_difference()** e **symmetric_difference_update()** che vanno ad escludere completamente i duplicati non riportandoli neanche una volta, a differenza di union() e update(). Cominciamo con symmetric_difference_update():

```
x = {"Milano", "Venezia", "Roma"}
y ={"Napoli", "Roma"}

x.symmetric_difference_update(y)
print(x)
```

```
{'Milano', 'Napoli', 'Venezia'}
```

Passiamo a symmetric_difference():

```
x = {"Milano", "Venezia", "Roma"}
y ={"Napoli", "Roma"}
z = x.symmetric_difference(y)

print(z)
```

```
{'Milano', 'Napoli', 'Venezia'}
```

Ancora una volta la differenza tra i due è che con symmetric_difference() viene creato un nuovo set mentre con symmetric_difference_update() verrà aggiornato il set già creato.

LEZIONE 16

DICTIONARY:

I dictionary sono collezioni di dati ordinati, modificabili ma che non permettono duplicati. Per creare un dictionary (dict) bisogna usare le parentesi graffe come per i set ma a differenza dei set bisogna creare una chiave ed un oggetto (come in JavaScript). Quindi, sintetizzando, i dict sono un insieme di coppie chiave-valore. Ad esempio, una coppia chiave-valore potrebbe essere nome (chiave) e Luca (valore). Creiamo un dict di esempio:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}
```

Come possiamo vedere abbiamo creato una dict che contiene 3 chiavi (nome, cognome, età) e 3 valori (Luca, Rossi, 25). Se vorremmo creare altri 3 valori, ad esempio Sara, Verdi, 36, non c'è bisogno di creare altre 3 chiavi ma si fruttano quelle già create avendo quindi introdotto delle proprietà come per la programmazione ad oggetti.

Stampiamo persona:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}
print(persona)
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25}
```

Vediamo ora come non sono ammesse chiavi duplicate, infatti se duplichiamo una chiave verrà stampata solo l'ultima inserita e l'altra è come se non esistesse.

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "eta": 36
}
print(persona)
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 36}
```

Come per le altre collezioni di dati possiamo usare il **type()** ed il **len()**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "eta": 36
}
print(type(persona))
print(len(persona))
```

```
<class 'dict'>
3
```

Vediamo un po' di **metodi per accedere agli elementi**. Il primo metodo sono le parentesi quadre **[]** con all'interno non l'indice ma la chiave.

```
main.py > ...
1   persona = {
2       "nome": "Luca",
3       "cognome": "Rossi",
4       "eta": 25,
5       "eta": 36
6   }
7   print(persona[])
                "cognome"        Dictionary key
                "eta"
                "nome"
```

Non appena mettiamo [] vicino a persona ci vengono suggerite le chiavi a cui poter accedere. Ne selezioniamo una e la vediamo a schermo.

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "eta": 36
}
print(persona["cognome"])
```

```
Rossi
```

Un altro metodo è il **get()**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "eta": 36
}
print(persona.get("nome"))
```

```
Luca
```

Se vogliamo prendere tutte le chiavi usiamo il metodo **keys()** che ci restituirà una lista con dentro le chiavi della persona:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "eta": 36
}
print(persona.keys())
```

```
dict_keys(['nome', 'cognome', 'eta'])
```

Usiamo invece **values()** per prendere la lista degli attributi. In questo esempio ho voluto creare una variabile, assegnargli values() e mandarla a schermo, per diversificare un po':

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

x = persona.values()

print(x)
```

```
dict_values(['Luca', 'Rossi', 25])
```

Usando **items()** avremo a schermo una lista di tuple con all'interno di ogni tupla l'accoppiata chiave-valore

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

x = persona.items()

print(x)
```

```
dict_items([('nome', 'Luca'), ('cognome', 'Rossi'), ('eta', 25)])
```

Per **controllare se una chiave esiste**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

print("nome" in persona)
```

```
True
```

Per **modificare un attributo con []** basta inserire nelle parentesi quadre la chiave da modificare ed assegniargli il nuovo valore

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

persona["nome"] = "Marco"

print(persona)
```

```
{'nome': 'Marco', 'cognome': 'Rossi', 'eta': 25}
```

In alternativa possiamo usare **update()**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

persona.update({"nome": "Anna"})

print(persona)
```

```
{'nome': 'Anna', 'cognome': 'Rossi', 'eta': 25}
```

Per **aggiungere un elemento** si possono usare le **[]** come per la modifica:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

persona["colore"] = "rosso"

print(persona)
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25, 'colore': 'rosso'}
```

Allo stesso modo usiamo update():

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

persona.update({"colore": "blu"})

print(persona)
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25, 'colore': 'blu'}
```

Per rimuovere gli elementi ci sono diversi metodi. Il primo è **pop()** con all'interno la chiave da rimuovere:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

persona.pop("nome")

print(persona)
```

```
{'cognome': 'Rossi', 'eta': 25}
```

Per rimuovere l'ultimo elemento si usa **popitem()**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

persona.popitem()

print(persona)
```

```
{'nome': 'Luca', 'cognome': 'Rossi'}
```

Per avere un dict vuoto si usa **clear()**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

persona.clear()

print(persona)
```

```
{}
```

Infine, per cancellare del tutto il dict creato si usa **del**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

del persona

print(persona)
```

```
NameError: name 'persona' is not defined
```

Si può eliminare anche solo una chiave e non tutto il dict:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

del persona["nome"]

print(persona)
```

```
{'cognome': 'Rossi', 'eta': 25}
```

Vediamo come ciclare gli elementi. Primo esempio:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

for x in persona:
    print(x)
```

```
nome
cognome
eta
```

in questo caso abbiamo a schermo la chiave. Se invece volevamo il valore:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

for x in persona:
    print(persona[x])
```

```
Luca
Rossi
25
```

Altri metodi sono **keys()** e **values()** che mandano a schermo rispettivamente le chiavi ed i valori:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

for x in persona.keys():
    print(x)
# manda a schermo le chiavi

print()

for x in persona.values():
    print(x)
# manda a schermo i valori
```

```
nome
cognome
eta

Luca
Rossi
25
```

Possiamo utilizzare **items()** ma dobbiamo ricordare che items genera una tupla per cui dobbiamo ciclare utilizzando 2 variabili:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

for x, y in persona.items():
    print(x, y)
```

```
nome Luca
cognome Rossi
eta 25
```

Per copiare si può usare il metodo **copy()** o il costruttore **dict**:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

x = persona.copy()

print(x)
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25}
```

Facendo copy() andiamo a creare una copia che se anche modificata non intacca persona, mentre se scrivevamo solo x = persona andavamo a copiare le referenze e modificando un valore in x lo modificavamo anche in persona. Stesso discorso vale per dict:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

x = dict(persona)

print(x)
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25}
```

Per concludere vediamo i **dict annidati**, ovvero dict in altri dict. Vediamo un esempio:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "indirizzo": {
        "citta": "Milano",
        "cap": "00000",
        "civico": 45
    }
}

print(persona)
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25, 'indirizzo': {'citta': 'Milano',
'cap': '00000', 'civico': 45}}
```

Per **prendere un dict dentro un dict** possiamo fare:

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "indirizzo": {
        "citta": "Milano",
        "cap": "00000",
        "civico": 45
    }
}

print(persona["indirizzo"])
# prendiamo tutto quello che c'è in indirizzo

print()
# lasciamo uno spazio vuoto

print(persona["indirizzo"]["civico"])
# prendiamo solo il civico che si trova in indirizzo
```

```
{'citta': 'Milano', 'cap': '00000', 'civico': 45}

45
```

LEZIONE 17

LE FUNZIONI:

Le funzioni si possono definire un blocco di codice riutilizzabile al cui interno ci sono delle istruzioni che verranno eseguite ogni volta che verranno richiamate. Per **creare una funzione** bisogna definirla usando def. Vediamo un esempio:

```
def fai_la_pasta():
```

Creata la funzione (importante non dimenticare le parentesi tonde) possiamo includere al suo interno i codici da eseguire. In maniera semplice mandiamo a schermo alcuni indicatori per la funzione fai_la_pasta():

```
def fai_la_pasta():
    print("metti l'acqua")
    print("fai bollire")
    print("metti la pasta")
```

Attenzione. In questo momento abbiamo solo creato la funzione e messo le istruzioni al suo interno, ma se vogliamo stampare tutto quello che c'è all'interno bisogna richiamarla. Per richiamare la funzione basta semplicemente andare nel programma e scrivere il nome della funzione.

```
def fai_la_pasta():
    print("metti l'acqua")
    print("fai bollire")
    print("metti la pasta")

fai_la_pasta()
# funzione richiamata. Ora possiamo vedere a schermo le istruzioni della funzione
```

```
metti l'acqua
fai bollire
metti la pasta
```

La funzione può essere richiamata quanto volte si vuole. Vediamo praticamente con un banale esempio:

```
def fai_la_pasta():
    print("metti l'acqua")
    print("fai bollire")
    print("metti la pasta")

fai_la_pasta()
fai_la_pasta()
fai_la_pasta()
```

```
metti l'acqua
fai bollire
metti la pasta
metti l'acqua
fai bollire
metti la pasta
metti l'acqua
fai bollire
metti la pasta
```

In questo banale esempio si può notare che è bastato mandare a schermo 3 volte (3 righe di comando) la funzione fai_la_pasta() ed abbiamo ottenuto 9 istruzioni. Se non era stata creata la funzione fai_la_pasta() dovevamo scrivere 9 print (9 righe di comando) per avere a schermo le 9 istruzioni.

Andiamo ora a **definire dei parametri** che ci permetteranno di prendere degli argomenti nella funzione. Ad esempio possiamo definire il tipo di pasta da mettere. Otterremo ora:

```
def fai_la_pasta(tipo_pasta):
    print("metti l'acqua")
    print("fai bollire")
    print("metti " + tipo_pasta)

fai_la_pasta("fusilli")
```

Ora, quando si va a richiamare la funzione fai_la_pasta() dobbiamo obbligatoriamente immettere al suo interno un argomento, che nel nostro caso è tipo_pasta. Così facendo possiamo richiamare sempre la stessa funzione ma possiamo scegliere anche l'argomento in base a quello che ci serve. Se non mettiamo nessun argomento otterremo un errore a schermo dato che nella nostra funzione l'argomento è stato dichiarato. Nel nostro esempio potremmo mettere una volta fusilli ed una volta spaghetti.

```
def fai_la_pasta(tipo_pasta):
    print("metti l'acqua")
    print("fai bollire")
    print("metti " + tipo_pasta)

fai_la_pasta("fusilli")
print()
fai_la_pasta("spaghetti")
```

```
metti l'acqua
fai bollire
metti fusilli

metti l'acqua
fai bollire
metti spaghetti
```

Da notare la **differenza tra parametri ed argomenti**. Il parametro è quello che definiamo nella funzione, nel nostro caso tipo_pasta, l'argomento è quello che effettivamente andremo a mettere quando richiamiamo la funzione, nel nostro esempio spaghetti e fusilli. Ampliamo l'esempio mettendo 2 parametri. Ad esempio potremmo volere la pasta in bianco o con il sugo. Potremmo allora mettere un parametro booleano che ci indichi False se non vogliamo il sugo e True se vogliamo il sugo. Per preparare la pasta in bianco mettiamo come secondo argomento False:

```
def fai_la_pasta(tipo_pasta, metti_sugo):
    print("metti l'acqua")
    print("fai bollire")
    print("metti " + tipo_pasta)
    if metti_sugo:
        print("prepara il sugo")

fai_la_pasta("fusilli", False)
```

```
metti l'acqua
fai bollire
metti fusilli
```

Per preparare la pasta con il sugo metteremo come secondo argomento True:

```
def fai_la_pasta(tipo_pasta, metti_sugo):
    print("metti l'acqua")
    print("fai bollire")
    print("metti " + tipo_pasta)
    if metti_sugo:
        print("prepara il sugo")

fai_la_pasta("fusilli", True)
```

```
metti l'acqua
fai bollire
metti fusilli
prepara il sugo
```

Se non si sa quanti di quanti parametri avremo bisogno nella funzione si usano gli **Arbitrary Arguments**, ovvero inserire un solo parametro generico preceduto da asterisco, ad esempio **opzioni*, per poi poterlo richiamare quando ci servono i vari argomenti. Se si rende conto che serviranno 2 argomenti allora gli stessi verranno richiamati scrivendo opzioni[0] e opzioni[1] rispettivamente per il 1° ed il 2° argomento. Vediamo un esempio:

```
def fai_la_pasta(*opzioni):
    print("metti l'acqua")
    print("fai bollire")
    print("metti " + opzioni[0])
    if opzioni[1]:
        print("prepara sugo")

fai_la_pasta("fusilli", True)
```

```
metti l'acqua
fai bollire
metti fusilli
prepara sugo
```

Quindi il numero di *opzioni* è arbitrario, definito in base alle esigenze ed è gestito con gli indici.

Si ha anche la possibilità di assegnare una chiave ed un valore alla funzione, in questo caso si parla di **Keyword Arguments**. I parametri si dichiarano nella funzione ma gli argomenti si assegnano quando la funzione viene richiamata. Vediamo un esempio:

```
def fai_la_pasta(tipo_pasta, sugo):
    print("metti l'acqua")
    print("fai bollire")
    print("metti "  +  tipo_pasta)
    if sugo:
        print("prepara sugo")

fai_la_pasta(tipo_pasta = "fusilli", sugo = True)
```

```
metti l'acqua
fai bollire
metti fusilli
prepara sugo
```

Combinando gli **Arbitrary Arguments** ed i **Keyword Arguments** si otterranno gli **Arbitrary Keyword Arguments** che però non verranno trattati in questo corso.

Vediamo ora i parametri di default. Sono dei parametri dichiarati direttamente nella funzione e quando si andrà a richiamare la funzione abbiamo la scelta di non mettere nessun argomento, e quindi verrà richiamato in automatico l'argomento inserito di default, oppure di inserire un argomento ed allora verrà modificato l'argomento di default. Per esempio se nella funzione dichiariamo metti_ pasta = spaghetti, quando andremo a richiamare la funzione se non mettiamo nessun argomento avremo come pasta spaghetti ma se mettiamo come argomento "fusilli" allora spaghetti verrà sostituito da fusilli.

```
def fai_la_pasta(tipo_pasta = "spaghetti"):
    print("metti l'acqua")
    print("fai bollire")
    print("metti " + tipo_pasta)
    print()

fai_la_pasta()
# di default viene messo l'argomento spaghetti

fai_la_pasta("fusilli")
# l'argomento di default spaghetti viene sostituito con fusilli
```

```
metti l'acqua
fai bollire
metti spaghetti

metti l'acqua
fai bollire
metti fusilli
```

Introduciamo ora il **return** nelle funzioni. Il return manda a schermo quelo che la funzione svolge. Vediamo un esempio:

```
def fai_somma(num1, num2):
    somma = num1 + num2

x = fai_somma(2, 4)

print(x)
```

```
None
```

In questo esempio abbiamo definito la funzione fai_somma() inserendo 2 parametri, num1 e num2. Al suo interno abbiamo fatto la somma dei 2. Poi abbiamo definito una variabile x = fai_somma ed abbiamo inseritoi 2 numeri da sommare ma quando mandiamo a schermo x non otteniamo nulla. Questo perché manca il return all'interno della funzione. Lo andiamo a mettere ed otteniamo a schermo la somma:

```
def fai_somma(num1, num2):
    somma = num1 + num2
    return somma

x = fai_somma(2, 4)

print(x)
```

```
6
```

In pratica sono entrati dei valori nella funzione, non sono più valori fissi come per la stringa, li abbiamo sommati e li abbiamo restituiti in uscita con il return.

LEZIONE 18

INTRODUZIONE ALLA PROGRAMMAZIONE AD OGGETTI: CLASSI E OGGETTI:

La programmazione ad oggetti si può definire come quel tipo di programmazione che va a prendere oggetti ed entità del mondo reale e le trasforma in modo programmatico in entità nel programma. Per capirlo meglio cominciamo con il creare una **classe** persona:

```
class persona:
    nome = "Luca"
    cognome = "Rossi"
```

Ora possiamo creare uno o più **oggetti** persona (persona1, persona2, ecc…) che avranno come parametri nome e cognome che andremo a definire per ogni persona.

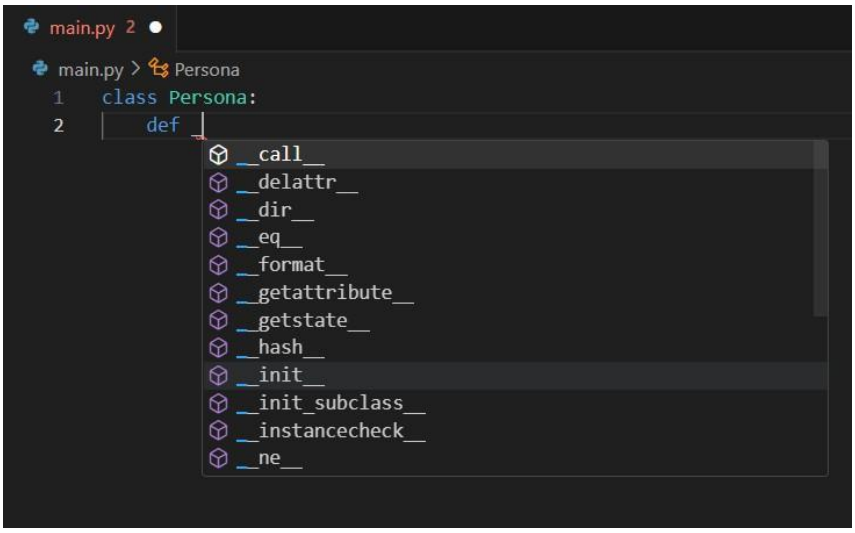
```
class Persona:
    nome = "Luca"
    cognome = "Rossi"

persona1 = Persona()
persona2 = Persona()
```

in questo momento abbiamo 2 persone diverse create dalla stessa classe. In pratica è come se avremo una stampante che stampa 2 volte la stessa cosa. In gergo possiamo dire che persona1 è un oggetto che deriva dall'**istanza** Persona(). L'esempio di sopra è poco pratico e poco realistico perché, se mandiamo a schermo nome o cognome (o entrambi) di persona1 saranno uguali a quelli di persona2.

```
class Persona:
    nome = "Luca"
    cognome = "Rossi"

persona1 = Persona()
persona2 = Persona()

print(persona1.nome)
print(persona2.nome)
```

```
Luca
Luca
```

E’ chiaramente improbabile che tutte le persone nascano con il nome di Luca Rossi, per cui introduciamo il concetto di **costruttore**.



Selezioniamo _init_

```
class Persona:
    def __init__(self) -> None:
    # def è una funzione
    # init ci completa
    # self è il riferimento a se stesso

persona1 = Persona()
persona2 = Persona()

print(persona1.nome)
print(persona2.nome)
```

self farà riferimento a persona1 se richiamato in persona1 e farà riferimento a persona2 se richiamato in persona2. Avendo self dovremo avere anche dei parametri. Nel nostro esempio nome e cognome. Avremo dunque:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome
```

Ora, quando si va a richiamare una persona(istanza) automaticamente avremo il costruttore che vuole i due parametri (nome e cognome).

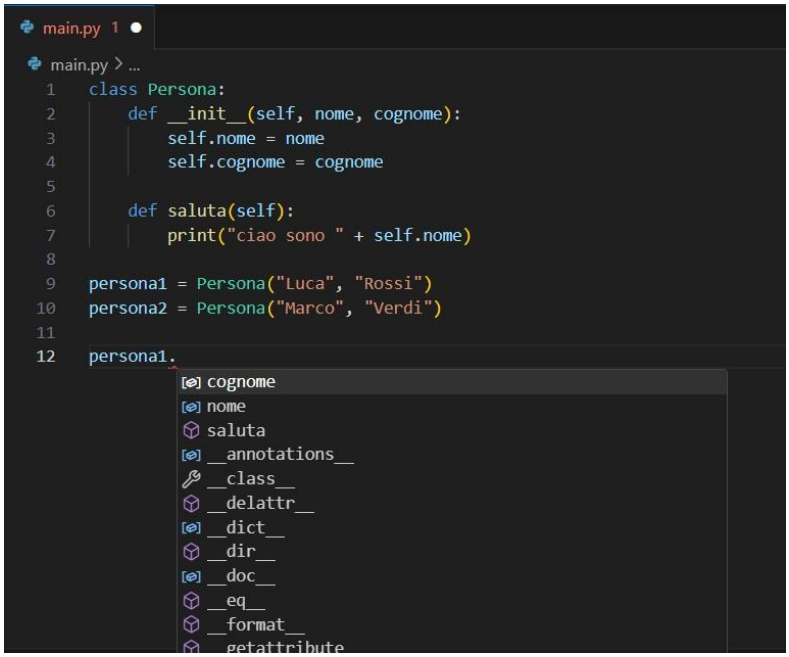
```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

persona1 = Persona("Luca", "Rossi")
persona2 = Persona("Marco", "Verdi")
```

In questo caso avremo 2 oggetti di istanza Persona (anche se sono 2 istanze diverse di Persona). Se vogliamo mandare in stampa il nome di una o dell'altra persona basta fare:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

persona1 = Persona("Luca", "Rossi")
persona2 = Persona("Marco", "Verdi")

print(persona1.nome)
print(persona2.nome)
```

```
Luca
Marco
```

Definita la classe Persona introduciamo i metodi all'interno della classe. Ad esempio, potremmo implementare un metodo saluta:



Una volta scritto il metodo saluta() basta scrivere *persona1.* ed in automatico ci compaiono gli attributi (o proprietà) ed i metodi di persona1. Richiamiamo il metodo saluta e mandiamo a schermo:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

persona1 = Persona("Luca", "Rossi")
persona2 = Persona("Marco", "Verdi")

persona1.saluta()
```

```
ciao sono Luca
```

Ovviamente anche per persona2 valgono le stesse proprietà e gli stessi metodi.

Vediamo come **modificare i parametri di un oggetto**:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

persona1 = Persona("Luca", "Rossi")
persona2 = Persona("Marco", "Verdi")

persona1.nome = "Maria"

persona1.saluta()
```

```
ciao sono Maria
```

Per **eliminare le proprietà di un oggetto**:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

persona1 = Persona("Luca", "Rossi")
persona2 = Persona("Marco", "Verdi")

del persona1.nome

persona1.saluta()
```

```
AttributeError: 'Persona' object has no attribute 'nome'
```

Per **eliminare tutto l'oggetto** basta fare *del* oggetto senza specificare gli attributi:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

persona1 = Persona("Luca", "Rossi")
persona2 = Persona("Marco", "Verdi")

del persona1

persona1.saluta()
```

```
NameError: name 'persona1' is not defined. Did you mean: 'persona2'?
```

LEZIONE 19

EREDITARIETÀ:

L'ereditarietà è un pilastro della programmazione ad oggetti e riguarda tutto ciò che le classi che derivano da altre classi ereditano. Facciamo un esempio creando una classe madre che chiameremo Persona ed una classe figlia, che chiameremo Insegnante, che deriverà da persona. Quindi la classe Insegnante estende Persona, nel senso che ne estende le proprietà ed i metodi.

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    pass
```

La parola pass è necessaria e serve per far capire che Insegnante può ereditare tutti i metodi di Persona. Se non si mette si avrà un errore. Andiamo a creare ora un insegnante ed una persona:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    pass

persona1 = Persona("Marco", "Rossi")
insegnante1 = Insegnante("Anna", "Neri")

print(persona1.cognome)
print(insegnante1.cognome)
```

```
Rossi
Neri
```

Notiamo subito, mandando in stampa i cognomi di persona1 ed insegnante1, che per la classe persona riusciamo a stampare il cognome perché all'interno abbiamo degli appositi metodi mentre per Insegnante, pur non avendo nulla, riusciamo a stampare comunque il cognome in quanto eredita metodi, proprietà e costruttori dalla classe madre Persona.

È importante sapere che, se nella classe figlia viene scritta la stessa funzione della classe madre ma con parametri diversi vale il principio della sovrascrittura, ovvero prevale la funzione della classe figlia sulla madre. Per evitare questo e mantenere i parametri della classe persona ed aggiungerne di extra in Insegnante si usa la **funzione super**, come vediamo nell'esempio sotto:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    def___init__(self, nome, cognome):
        super().__init__(nome, cognome) # nome e cognome ereditate da Persona
```

Avendo inserito dei costruttori non serve più il pass. Ora andiamo a vedere le **proprietà esclusive** di Insegnante, ovvero **della classe figlia**:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    def___init__(self, nome, cognome, materia): # materia è esclusivo di Insegnante
        super().__init__(nome, cognome) # nome e cognome ereditate da Persona
        self.materia = materia
```

Ricapitolando, in Insegnante si hanno 4 argomenti (self, nome, cognome, materia) mentre in super se ne hanno 3 (self, nome, cognome) che sono quelli della madre e che non possono essere modificabili con l'aggiunta o rimozione di altri argomenti. Gli argomenti si possono aggiungere solo e soltanto nel costruttore della classe figlia e non in super.

Mandiamo ora in stampa la materia di insegnante1:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    def_init_(self, nome, cognome, materia): # materia è esclusivo di Insegnante
        super()._init_(nome, cognome) # nome e cognome ereditate da Persona
        self.materia = materia

persona1 = Persona("Marco", "Rossi")
insegnante1 = Insegnante("Anna", "Neri", "Informatica")

print(insegnante1.materia)
```

```
Informatica
```

Vediamo che correttamente viene restituito “Informatica”. Se invece mandiamo in stampa la materia di persona1 avremo un errore perché persona1 non ha come argomento *materia*.

Andiamo ora a vedere i **metodi e l’ovverriding**. Intanto possiamo introdurre il concetto facendo un esempio del metodo saluta applicato ad Insegnante.

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    def_init_(self, nome, cognome, materia): # materia è esclusivo di Insegnante
        super()._init_(nome, cognome) # nome e cognome ereditate da Persona
        self.materia = materia

persona1 = Persona("Marco", "Rossi")
insegnante1 = Insegnante("Anna", "Neri", "Informatica")

insegnante1.saluta()
```

```
ciao sono Anna
```

Però si sa che un insegnante in classe deve salutare in maniera diversa da quando è a casa ecc… per cui definiamo un altro metodo saluta nella classe figlia Insegnante che sostituirà quello della classe madre:

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    def_init_(self, nome, cognome, materia): # materia è esclusivo di Insegnante
        super()._init_(nome, cognome) # nome e cognome ereditate da Persona
        self.materia = materia

    def saluta(self):
        print("Buongiorno sono " + self.nome + " " + self.cognome)

persona1 = Persona("Marco", "Rossi")
insegnante1 = Insegnante("Anna", "Neri", "Informatica")

insegnante1.saluta()
```

```
Buongiorno sono Anna Neri
```

In questo esempio si è fatto l'ovverride del metodo saluta.

Ovviamente in insegnante1 nessuno di mettere nuovi metodi oltre a quelli dell'ovveride.

```
class Persona:
    def_init_(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome
    def saluta(self):
        print("ciao sono " + self.nome)

class Insegnante(Persona):
    def_init_(self, nome, cognome, materia): # materia è esclusivo di Insegnante
        super()._init_(nome, cognome) # nome e cognome ereditate da Persona
        self.materia = materia
    def saluta(self):
        print("Buongiorno sono " + self.nome + " " + self.cognome)
    def dai_voto(self):
        print("Ottimo compito, 8")

persona1 = Persona("Marco", "Rossi")
insegnante1 = Insegnante("Anna", "Neri", "Informatica")

insegnante1.saluta()
insegnante1.dai_voto()
```

```
Buongiorno sono Anna Neri
Ottimo compito, 8
```

LEZIONE 20

LO SCOPE DELLE VARIABILI:

Lo scope è la disponibilità della variabile nel codice. Come sappiamo si hanno variabili nelle funzioni e variabili fuori dalle funzioni, lo scope è quella porzione di codice dove la variabile è disponibile.

```
x = 300          # variabile al di fuori di una funzione

def funzione():
    y = 100      # variabile dentro la funzione
```

Lo scope può essere **scope locale** e **scope globale**. Quello locale è all'interno di una funzione. Vediamo un esempio concreto:

```
def funzione():
    x = 100
    print(x)

print(x)
```

```
NameError: name 'x' is not defined
```

Pur mandando a schermo x otterremo un errore perché la variabile x è definita solo a livello locale nella funzione per cui a livello globale non esiste. Se vogliamo mandarla in stampa dobbiamo stampare la funzione:

```
def funzione():
    x = 100
    print(x)

funzione()
```

```
100
```

Però potremmo prendere il valore locale e stamparlo normalmente furi dalla funzione usando il return:

```
def funzione():
    x = 100
    print(x)
    return x

x = funzione()

print(x)
```

```
100
100
```

A schermo si ha sia il 100 della funzione (il locale) che il 100 del print(x) (il globale). Altra cosa importante per lo scope locale è che si possono chiamare le sotto funzioni all'interno delle funzioni:

```
def funzione():
    x = 100
    def sottofunzione(): # definiamo una sottofunzione
        print(x)
    sottofunzione() # serve per dire che in funzione c'è sia x che la sottofunzione

funzione() # manda in stampa
```

```
100
```

Lo scope locale è invece quello che si ha a disposizione per tutto il programma.

```
x = 300

def funzione():
    x = 100
    def sottofunzione():
        print(x)
    sottofunzione()

funzione()

print(x)
```

```
100
300
```

Come notiamo prima viene stampata la x locale (100) e dopo la x globale (300) che sono due variabili totalmente diverse. È consigliabile, dunque, usare sempre nomi diversi anche per variabili globali e locali.

Se si vuole utilizzare anche in locale la x globale si usa la parola global (**keyword global)**:

```
x = 300

def funzione():
    global x    # x prende il valore globale 300
    print(x)

funzione()

print(x)
```

```
300
300
```

Se viene modificato il valore di x in locale in automatico verrà cambiato anche il valore della x globale grazie alla parola global.

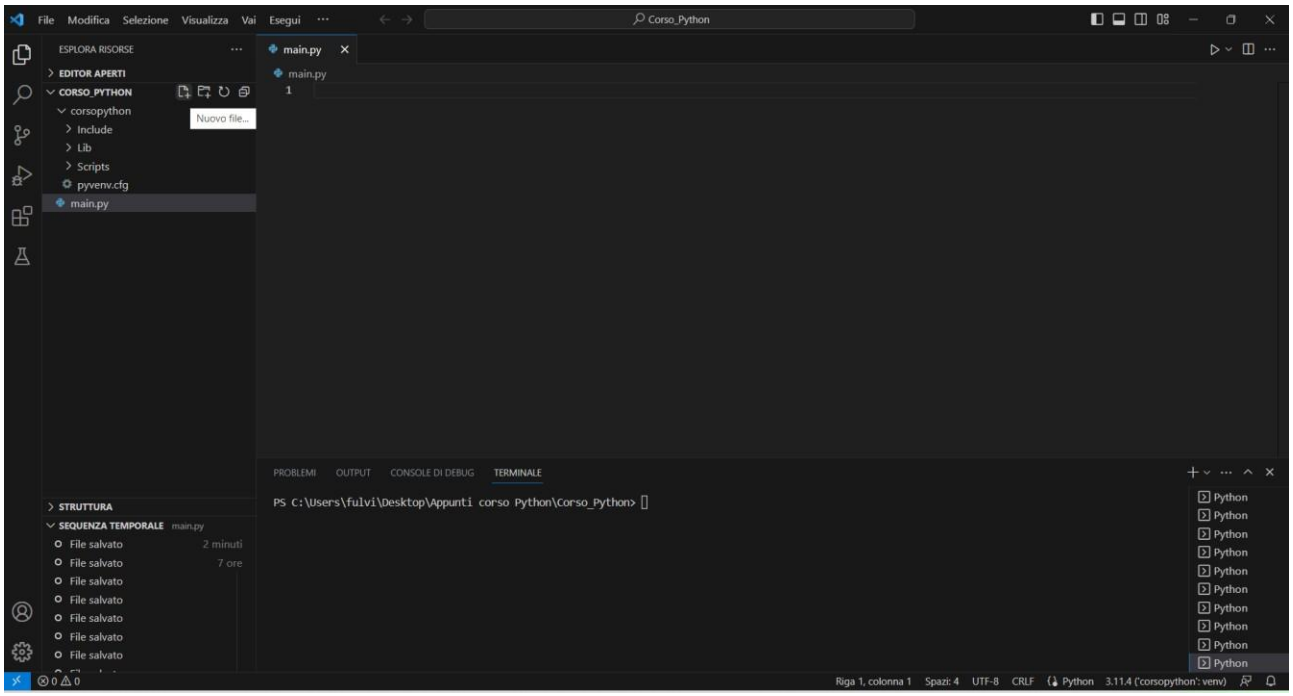
```
x = 300

def funzione():
    global x    # x prende il valore globale 300
    x = 100
    print(x)

funzione()

print(x)
```

```
100
100
```

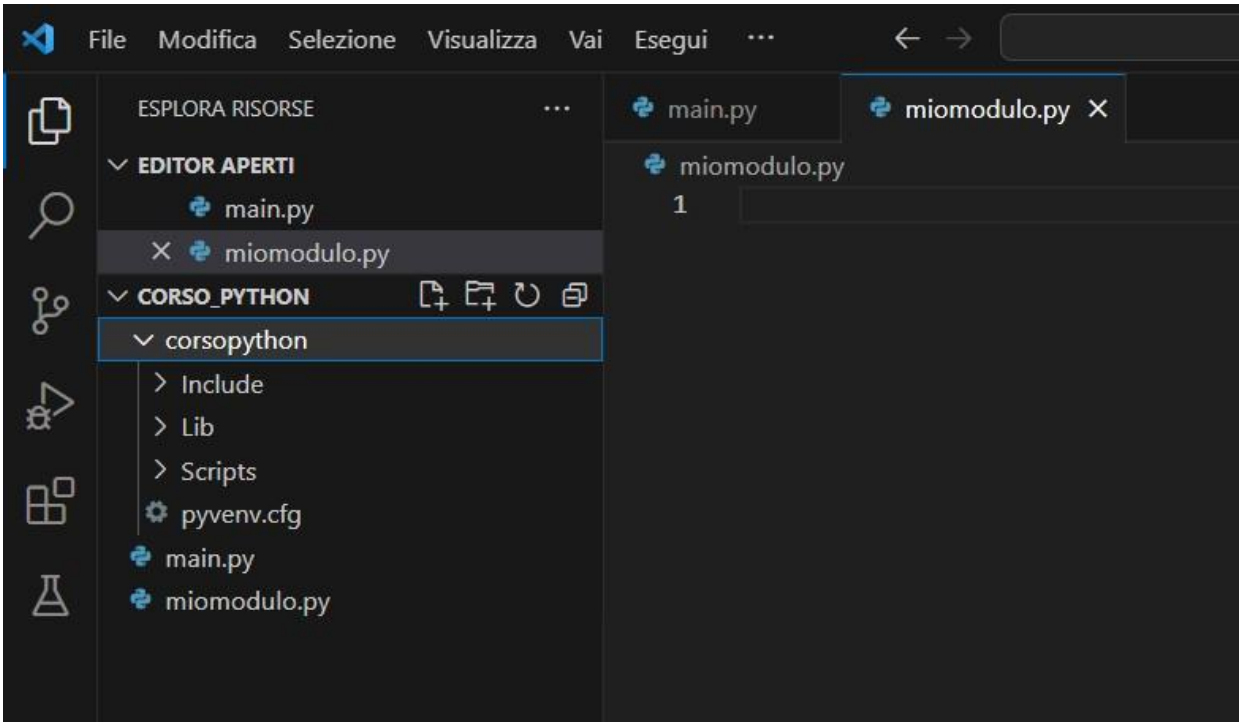
LEZIONE 21

MODULI:

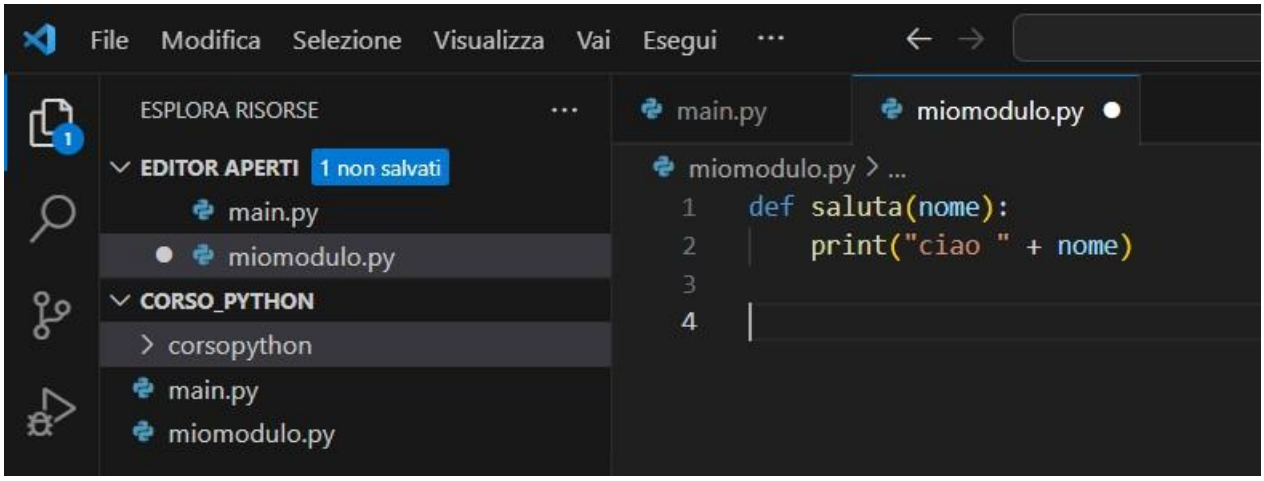
I moduli sono come delle librerie, ovvero file esterni con delle funzioni (o anche variabili) che vogliamo includere nel programma. Per creare un modulo andare a sinistra sotto la cartella CORSO_PYTHON e premere sulla prima icona, nuovo file:



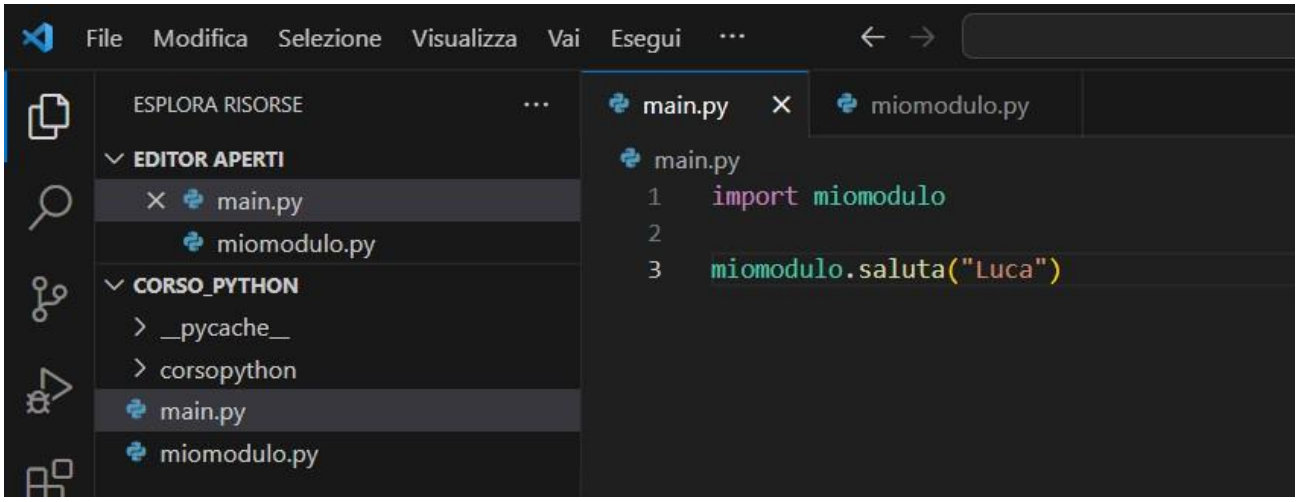
Fatto ciò, nominiamo il modulo come **miomodulo.py**:



All’interno del modulo definiamo una funzione saluta():



Per includere il modulo appena creato basta semplicemente andare nel main ed importarlo:




```
import miomodulo

miomodulo.saluta("Luca")
```

```
ciao Luca
```

Quindi, come si evince dall’esempio, un modulo è un file contenete da una a infinite funzioni che possono essere utilizzate (importate) in base a ciò che serve. Allo stesso modo possiamo importare anche le variabili. Andiamo in miomodulo e facciamo un dict persona:

miomodulo:

```
def saluta(nome):
    print("ciao " + nome)

persona1 = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}
```

Nel **main** invece possiamo stampare qualunque cosa di persona1:

```
import miomodulo

x = miomodulo.persona1["nome"] # creiamo una variabile x per passargli il nome di
persona1

miomodulo.saluta(x)
```

```
ciao Luca
```

Una cosa importante, da tener ben presente, è salvare ogni qual volta si passa da un modulo ad un altro (ad esempio da miomodulo a main) altrimenti il programma potrebbe restituire errori che in realtà non ci sono, come ad esempio la mancanza di un attributo che in realtà è stato appena creato ma non essendo salvato il programma non lo riconosce.

Ora vediamo come creare un **alias**. L'alias può servire quando ci sono nomi molto lunghi per i moduli e si vuole abbreviare un po'. In genere si tende a chiamare i moduli con 2 o 3 lettere al massimo. Nel nostro caso potremmo chiamare miomodulo con la sigla em, ad esempio. Per fare ciò si scrive:

```
import miomodulo as em

x = em.persona1["nome"] # creiamo una variabile x per passargli il nome di persona1

em.saluta(x)
```

```
ciao Luca
```

Vediamo che otteniamo lo stesso risultato. Importante tenere presente che una volta messo l'alias potremmo usare solo quello e non più il nome per esteso.

Python ha una serie di moduli già definiti al suo interno. Ad esempio, c'è il **modulo platform** che fa vedere le caratteristiche della piattaforma che si utilizza, come ad esempio ambiente di lavoro, processore ecc…:

```
import platform

print(platform.processor())
print(platform.system())
print(platform.architecture())
```

```
AMD64 Family 23 Model 104 Stepping 1, AuthenticAMD
Windows
('64bit', 'WindowsPE')
```

Un modulo molto interessante è il **modulo math** che serve per le operazioni matematiche. Vediamo qualche esempio:

```
import math

print(math.floor(2.7)) # arrotonda per difetto
print(math.sin(0)) # calcola il seno
print(math.cos(0)) # calcola il coseno
```

```
2
0.0
1.0
```

Per vedere tutte le funzioni che ha un modulo possiamo usare la **funzione dir**. Ad esempio, vediamo cosa ha da offrire il modulo math:

```
import math

print(dir(math))
```

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'cbrt', 'ceil', 'comb', 'copysign',
'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'exp2', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow',
'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc', 'ulp']
```

Per **importare solo una parte del modulo** basta richiamare solo la parte del modulo che ci serve. Ad esempio, tornando al modulo miomodulo precedentemente creato abbiamo:

```
def saluta(nome):
    print("ciao " + nome)

persona1 = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}
```

Per importare solo persona1 nel main, e non tutto miomodulo, basta fare:

```
from miomodulo import persona1

print(persona1["nome"])
```

```
Luca
```

Ricordiamo che importiamo solo persona1 da miomodulo per cui quando andiamo il print(), miomodulo non è riconosciuto mentre sarà riconosciuto solo persona1.

Per sapere tutti i moduli di Python basta fare una ricerca e sapremo quali moduli si possono importare di default e quale funzione svolgono.

LEZIONE 22

DATA E ORA IN PYTHON CON DATETIME:

Python non ha un proprio tipo di data, quindi per lavorare con le date bisogna per forza importare il modulo datetime:

```
import datetime
```

Facciamo qualche esempio:

```
import datetime

x = datetime.datetime.now() # data attuale

print(x)
```

```
2023-08-29 23:16:42.306700
```

Abbiamo a schermo la data in formato inglese con anno/mese/giorno + ora/minuti/secondi/centesimi.

Creiamo una data partendo da una data predefinita:

```
import datetime

x = datetime.datetime(2012, 6, 28)

print(x)
```

```
2012-06-28 00:00:00
```

Abbiamo creato un oggetto data. Ora parliamo di **strftime (string format time)**. Con strftime prendiamo un oggetto data e lo convertiamo nel formato che più ci piace. Ad esempio:

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%B"))

# con il "%B" non restituirà la data attuale ma il mese attuale
```

```
August
```

Altre formattazioni della **lista dei parametri disponibili per la data in Python** sono elencate sotto:

- **`%d`:** Giorno del mese (01-31)
- **`%m`:** Mese (01-12)
- **`%Y`:** Anno (es. 2023)
- **`%H`:** Ore (00-23)
- **`%M`:** Minuti (00-59)
- **`%S`:** Secondi (00-59)
- **`%A`:** Nome del giorno (es. Lunedì)
- **`%B`:** Nome del mese (es. Gennaio)
- **`%Y-%m-%d %H:%M:%S`:** Formato esteso con data e orario

Ovviamente ce ne sono molte altre che si possono trovare facilmente online.

Tutti questi parametri possono essere combinati. Vediamo **esempi di formattazione data**:

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%d/%m/%Y"))

# %d restituisce il giorno
# / è il divisore che vogliamo usare
# %m restituisce il mese
# %Y restituisce l'anno
```

```
30/08/2023
```

Altro esempio:

```
import datetime

x = datetime.datetime.now()

print(x.strftime("%A %d %B %Y"))
```

```
Wednesday 30 August 2023
```

LEZIONE 23

IL MODULO MATH NEL DETTAGLIO – MATEMATICA AVANZATA:

Come già accennato in precedenza Python mette a disposizione il modulo Math per operazioni matematiche avanzate. Vediamo alcuni esempi più dettagliati rispetto alle lezioni precedenti. Intanto sappiamo già che in Python possiamo costruire delle semplici espressioni tipo quella sottostante:

```
x = 5
y = x + 7

z = (x+(y*3)/2)+3*y

print(z)
```

```
59.0
```

Introduciamo ora le **funzioni built-in**, cioè funzioni già costruite in Python tipo min, max, abs, pow, che non hanno bisogno di importare math:

```
x = 2
y = 3

print(min(x,y)) # stampa il minimo tra x e y
print(max(x,y)) # stampa il massimo tra x e y
print(abs(x)) # stampa il valore assoluto di x
print(pow(x,y)) # stampa x elevato ad y
```

```
2
3
2
8
```

Tuttavia, mancano alcune funzioni fondamentali, tipo la radice quadrata. Qui ci viene in aiuto il modulo math con le sue funzioni interne tipo sqrt, ceil, floor, pi. Vediamo alcuni esempi:

```
import math

x = math.sqrt(16)
print(x)
# ritorna la radice quadrata

z = math.pi
print(z)
# ritorna pi greco

y = 3.6
print(math.ceil(y))
# arrotonda per eccesso

print(math.floor(y))
# arrotonda per difetto
```

```
4.0
3.141592653589793
4
3
```

Esistono anche altre librerie per analisi dati tipo:

1. **Pandas -** set di strumenti più utili per esplorare, pulire e analizzare i dati
2. **Numpy -** utilizzato principalmente per il supporto di array N-dimensionali
3. **Scikit-learn -** per l'apprendimento automatico
4. **Gradio -** creare e distribuire app Web per modelli di machine learning in sole tre righe di codice
5. **Tensorflow -** implementazione di reti neurali
6. **Keras -** creare modelli di apprendimento profondo, in particolare reti neurali
7. **SciPy -** funzioni scientifiche e funzioni matematiche derivate da NumPy
8. **StatsModels -** libreria per fare statistiche hardcore. Questa libreria multifunzionale è una miscela di diverse librerie Python, che prende le sue caratteristiche grafiche e funzioni da Matplotlib, per la gestione dei dati, usa Pandas, per la gestione di formule R-like, usa Pasty ed è costruita su NumPy e SciPy.
9. **Plotly -** creare dashboard dinamici utilizzando visualizzazioni Plotly. Dash è un'interfaccia Python basata sul Web che elimina la necessità di JavaScript in questi tipi di applicazioni Web analitiche e consente di eseguire questi grafici online e offline.
10. **Seaborn -** seaborn è una libreria efficace per la creazione di diverse visualizzazioni. Una delle caratteristiche più importanti di Seaborn è la creazione di dati visivi amplificati.

LEZIONE 24

JSON:

JSON (JavaScript Object Notation) è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. JSON è un formato di testo completamente indipendente dal linguaggio di programmazione, ma utilizza convenzioni conosciute dai programmatori di linguaggi della famiglia del C, come C, C++, C#, Java, JavaScript, Perl, Python, e molti altri. Questa caratteristica fa di JSON un linguaggio ideale per lo scambio di dati.

Vediamo come **leggere JSON in Python**. Innanzi tutto, bisogna importare il modulo e poi creare una variabile di tipo stringa:

```
import json

x = '{"nome": "Luca", "cognome": "Rossi", "eta": 25}'
```

Utilizziamo il metodo json.loads ed otteniamo un dictionary di Python, che è la cosa più simile agli oggetti in Javascript:

```
import json

x = '{"nome": "Luca", "cognome": "Rossi", "eta": 25}'

y= json.loads(x)

print(y)

# otteniamo un dict a schermo
```

```
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25}
```

Se andiamo a vedere il tipo avremo la conferma che è un dictionary:

```
import json

x = '{"nome": "Luca", "cognome": "Rossi", "eta": 25}'

y= json.loads(x)

print(type(y))
```

```
<class 'dict'>
```

Quindi, riassumendo, da una stringa abbiamo letto un json con loads ed ottenuto un dict in Python.

Ora vediamo come **passare da Python a json**. Intanto creiamo un dict e poi con json.dumps per generare una stringa, quindi un json appunto:

```
import json

x = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

y = json.dumps(x)

print(y)        # stampiamo la stringa y (json)
print(type(y))  # stampiamo il tipo di y per conferma
```

```
{"nome": "Luca", "cognome": "Rossi", "eta": 25}
<class 'str'>
```

Riassumendo, per passare da Python a json abbiamo creato una dict e convertito la dict in stringa tramite dumps.

I **dati convertibili da Python in json** sono:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

Facciamo un esempio di tuple a json e di stringa a json, ma il discorso è uguale per tutti i tipi:

```
import json

y = json.dumps("Milano")
print(y)
print(type(y))

x = json.dumps(("Roma", "Firenze", "Venezia"))
print(x)
print(type(x))
```

```
"Milano"
<class 'str'>
["Roma", "Firenze", "Venezia"]
<class 'str'>
```

Vediamo come formattare il json. Iniziamo con il creare un dict contenente varie informazioni:

```
import json

x = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "isOnline": False,
    "interessi": ["calcio", "basket"],
    "monetineInTasca": 4.56,
    "fidanzata": None
}

y = json.dumps(x)
print(y)
```

```
{"nome": "Luca", "cognome": "Rossi", "eta": 25, "isOnline": false, "interessi":
["calcio", "basket"], "monetineInTasca": 4.56, "fidanzata": null}
```

Vediamo che abbiamo un jason contenete diverse informazioni che potrebbero risultare scomode da leggere su una sola riga. Per renderlo più leggibile formattiamo il json con indent:

```python
import json

x = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "isOnline": False,
    "interessi": ["calcio", "basket"],
    "monetineInTasca": 4.56,
    "fidanzata": None
}

y = json.dumps(x, indent=4)
print(y)
```

```
{
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "isOnline": false,
    "interessi": [
        "calcio",
        "basket"
    ],
    "monetineInTasca": 4.56,
    "fidanzata": null
}
```

Ora vediamo che la formattazione è decisamente migliorata.

Volendo si possono usare anche i separatori. Possiamo, ad esempio, separare riga con riga tramite un punto e separare la coppia chiave/valore tramite =

```
import json

x = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "isOnline": False,
    "interessi": ["calcio", "basket"],
    "monetineInTasca": 4.56,
    "fidanzata": None
}

y = json.dumps(x, indent=4, separators=(". ", "= "))
print(y)
```

```
{
    "nome"= "Luca".
    "cognome"= "Rossi".
    "eta"= 25.
    "isOnline"= false.
    "interessi"= [
        "calcio".
        "basket"
    ].
    "monetineInTasca"= 4.56.
    "fidanzata"= null
}
```

Infine, vediamo come **ordinare le json** usando sort_keys:

```
import json

x = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25,
    "isOnline": False,
    "interessi": ["calcio", "basket"],
    "monetineInTasca": 4.56,
    "fidanzata": None
}

y = json.dumps(x, indent=4, sort_keys=True)
print(y)
```

```
{
    "cognome": "Rossi",
    "eta": 25,
    "fidanzata": null,
    "interessi": [
        "calcio",
        "basket"
    ],
    "isOnline": false,
    "monetineInTasca": 4.56,
    "nome": "Luca"
}
```

LEZIONE 25

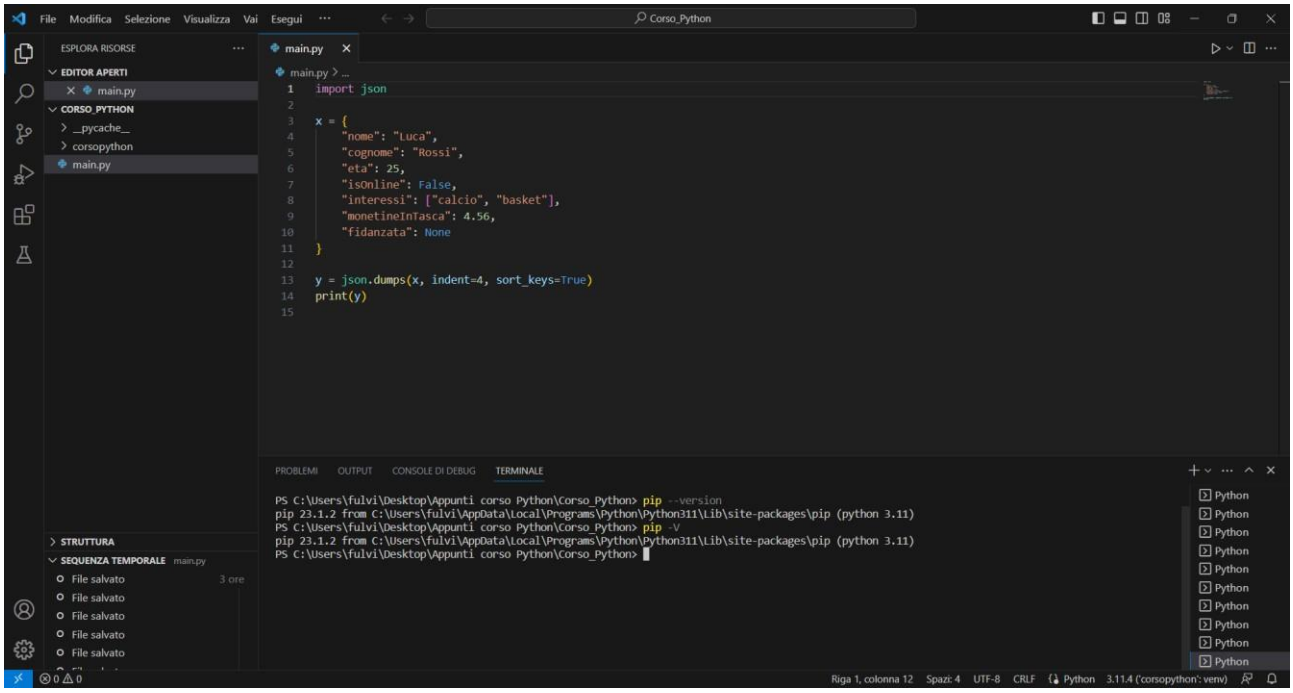
INSTALLARE NUOVI MODULI CON PIP:

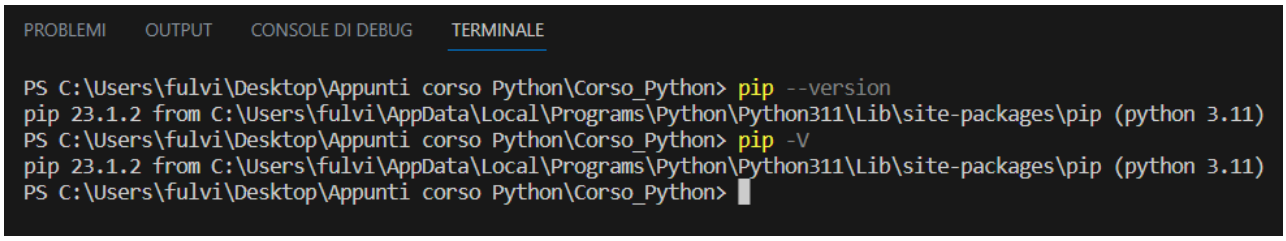
PIP è un package manager, ovvero un gestore di pacchetti con funzionalità extra, creati da terzi utenti, che si possono integrare, installare e disinstallare nel nostro programma. Un pacchetto si può definire fondamentalmente come un modulo che verrà richiamato quando serve. Quindi va installato il pacchetto tramite PIP e poi va importato come un normale modulo. Per prima cosa bisogna verificare se PIP è già installato. Per fare ciò andare in console e digitare

pip –version

oppure

pip -V





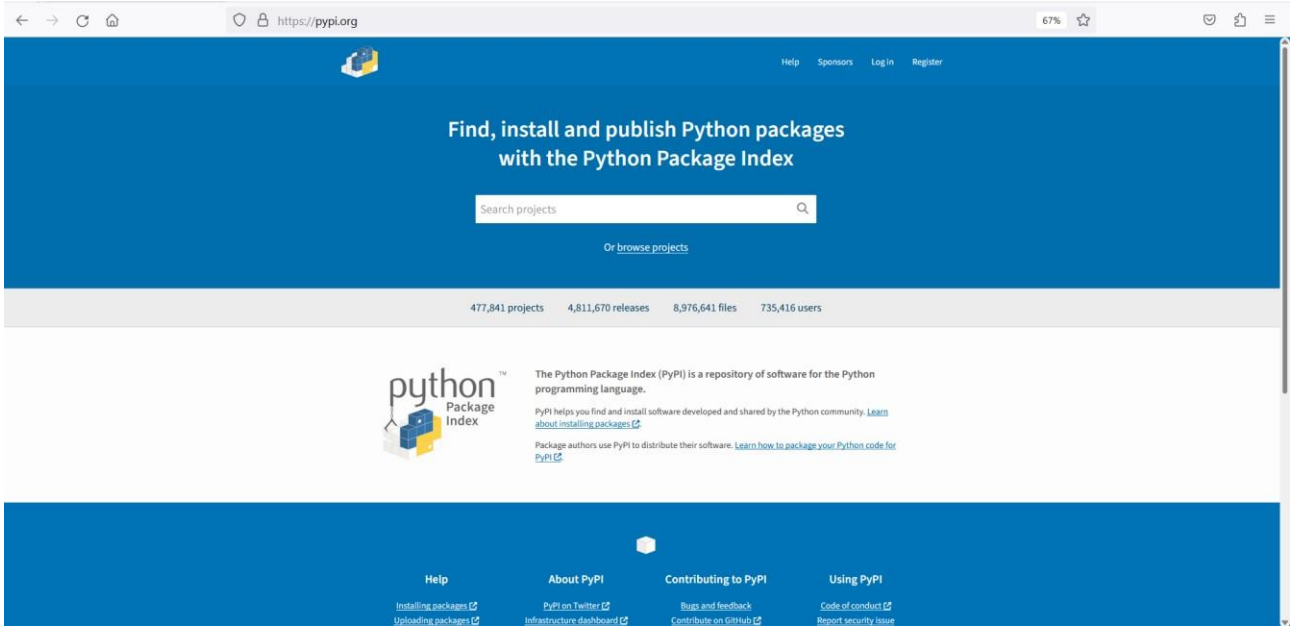
Nel mio caso risulta già installata la versione 23.1.2. Se non fosse installato basta andare in console e digitare il comando

python get-pip.py

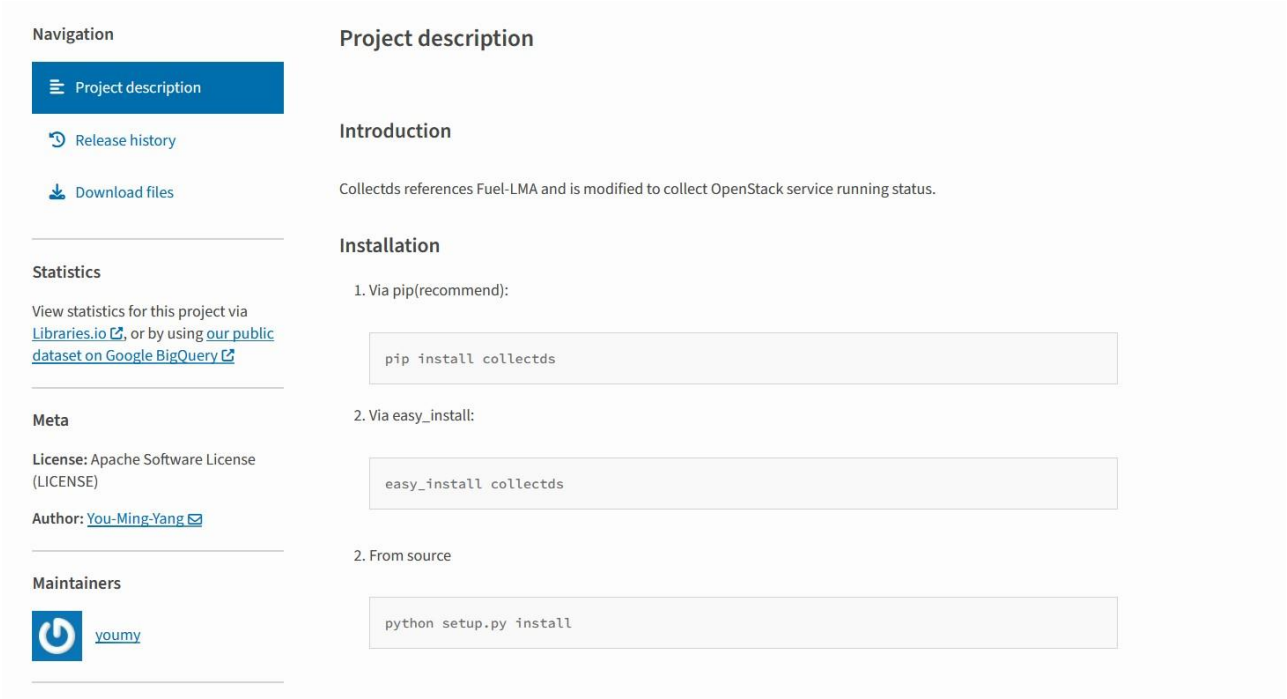
ed automaticamente verrà installato. Dopo l'installazione verificare l'avvenuto download di pip tramite uno dei due comandi visti in precedenza.

Una volta che si ha PIP si può procedere con il **download dei pacchetti desiderati**. Se non si hanno dei pacchetti già a disposizione se ne possono cercare molti su questo sito:

https://pypi.org



Tra i tanti pacchetti proposti ne troviamo uno che ci interessa e lo installiamo. Quando viene trovato il progetto che interessa la pagina sarà simile alla seguente



Nell'esempio di sopra si vuole installare ***collectds***. Allora basta andare nella sezione "Installation" e copiare la scritta ***pip install collectds*** nel terminale dei comandi di VSC ed il gioco è fatto.

Nel nostro caso vogliamo installare ***camel case***. Basta allora dare il comando ***pip install camel case***:

```
PROBLEMI    OUTPUT    CONSOLE DI DEBUG    TERMINALE

PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip --version
pip 23.1.2 from C:\Users\fulvi\AppData\Local\Programs\Python\Python311\Lib\site-packages\pip (python 3.11)
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip -V
pip 23.1.2 from C:\Users\fulvi\AppData\Local\Programs\Python\Python311\Lib\site-packages\pip (python 3.11)
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip install camelcase
```

Si inizieranno ad installare i pacchetti fino a quando non abbiamo il messaggio di installazione con successo:

```
PROBLEMI    OUTPUT    CONSOLE DI DEBUG    TERMINALE

PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip --version
pip 23.1.2 from C:\Users\fulvi\AppData\Local\Programs\Python\Python311\Lib\site-packages\pip (python 3.11)
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip -V
pip 23.1.2 from C:\Users\fulvi\AppData\Local\Programs\Python\Python311\Lib\site-packages\pip (python 3.11)
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip install camelcase
Collecting camelcase
  Downloading camelcase-0.2.tar.gz (1.3 kB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: camelcase
```

```
PROBLEMI    OUTPUT    CONSOLE DI DEBUG    TERMINALE

  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: camelcase
  Building wheel for camelcase (pyproject.toml) ... done
  Created wheel for camelcase: filename=camelcase-0.2-py3-none-any.whl size=1779 sha256=2c57cead740fe2f46de80bbb63d1c777477d7baa3cbd89c948c34dd78d
2440f7
  Stored in directory: c:\users\fulvi\appdata\local\pip\cache\wheels\9b\03\38\e96c3419390e51c6f282e9793e392eb2996b4059eb6739165b
Successfully built camelcase
Installing collected packages: camelcase
Successfully installed camelcase-0.2

[notice] A new release of pip is available: 23.1.2 -> 23.2.1
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python>
```

Installato il pacchetto, per **utilizzarlo** basta semplicemente **importarlo**:

```
import camelcase
```

Da questo momento è possibile utilizzare tutte le funzioni del pacchetto camelcase.

Per **disinstallare un pacchetto** basta scrivere nel terminale comandi:

pip uninstall camelcase

e alla domanda se siamo sicuri basta premere Yes (**Y**) ed il pacchetto sarà rimosso

```
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip uninstall camelcase
Found existing installation: camelcase 0.2
Uninstalling camelcase-0.2:
  Would remove:
    c:\users\fulvi\appdata\local\programs\python\python311\lib\site-packages\camelcase-0.2.dist-info\*
    c:\users\fulvi\appdata\local\programs\python\python311\lib\site-packages\camelcase\*
Proceed (Y/n)? y
  Successfully uninstalled camelcase-0.2
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python>
```

Per **verificare tutti i pacchetti installati** basta dare il comando

pip list

```
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python> pip list
Package    Version
---------- -------
pip        23.1.2
setuptools 65.5.0

[notice] A new release of pip is available: 23.1.2 -> 23.2.1
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\fulvi\Desktop\Appunti corso Python\Corso_Python>
```

LEZIONE 26

TRY, EXCEPT, FINALLY:

Durante la stesura di un programma possono capitare degli errori che fanno crashare il programma stesso (fatal error). In questo caso ci vengono in soccorso Try ed Except. Try permette di testare un blocco, Except permette di raccogliere l'errore che potrebbe essersi generato in Try mentre Finally permette di eseguire un altro tipo di codice una volta finiti Try ed Except.

Vediamo un esempio di Try. Scriviamo un semplice blocco di codice:

```
x = 5
print(x)
```

```
5
```

Nell'esempio non ci sono assolutamente problemi; ma se non dichiariamo la variabile x allora avremo un errore:

```
print(x)
```

```
NameError: name 'x' is not defined
```

In questo caso useremo Try per provare il codice ed Except per restituirci un messaggio una volta verificato che il codice è errato:

```
try:                # prova il codice
    print(x)
except:             # restituisce un messaggio in caso di codice errato
    print("c'è stato un errore")
```

```
c'è stato un errore
```

Se invece del messaggio di errore non vorremmo fare nulla ma semplicemente passare avanti basta usare pass:

```
try:
    print(x)
except:
    pass
```

Potremmo trovarci nel caso di **errori multipli** o **multiple eccezioni** e gestirle come sotto:

```
try:
    print(x)
except NameError:
    print("x non definita")     # restituisce questo messaggio solo nei casi di
NameError
except:
    print("non è name error")   # restituisce questo messaggio in tutti gli altri
case
```

```
x non definita
```

altro esempio:

```
x = 5
try:
    print(x + "ciao")
except NameError:
    print("x non definita")     # restituisce questo messaggio solo nei casi di
NameError
except:
    print("non è name error")   # restituisce questo messaggio in tutti gli altri
case
```

```
non è name error
```

Possiamo aggiungere anche else. Vediamo un esempio:

```
x = 5
try:
    print(x)
except:
    print("non è name error")
else:
    print("tutto per il verso giusto")
```

```
5
tutto per il verso giusto
```

In questo caso, non essendoci errori, stampa l'istruzione esatta e l'istruzione contenuta nell'else.

Andiamo ora a vedere finally. Finally esegui un'istruzione a prescindere se c'è l'errore o meno. Nel caso in cui non ci sia l'errore:

```
x = 5
try:
    print(x)
except:
    print("x non definita")
finally:
    print("finally restituito a prescindere")
```

```
5
finally restituito a prescindere
```

mentre se l'errore c'è si ha:

```
try:
    print(x)
except:
    print("x non definita")
finally:
    print("finally restituito a prescindere")
```

```
x non definita
finally restituito a prescindere
```

Vediamo un ultimo esempio per alzare/lanciare una eccezione (**raise/throw exception**):

```
x = -1

if x < 0:
    raise Exception("Numero minore di zero")

print(x) # questa istruzione non verrà eseguita perchè sopra c'è un'eccezione
```

```
Exception: Numero minore di zero
```

LEZIONE 27

INPUT DEGLI UTENTI – ESERCIZIO PRATICO:

Come già accennato ad inizio corso, per inserire dati da terminale di un utente basta usare input. Facciamo un semplice esempio dove inseriamo 2 numeri da tastiera ed il programma ci restituisce la somma.

```
x = input("inserisci numero 1: ")
y = input("inserisci numero 2: ")

somma = float(x) + float(y)

print("la somma è: " + str(somma))
```

```
inserisci numero 1: 3
inserisci numero 2: 7.4
la somma è: 10.4
```

Vediamo ora un esercizio complesso

Si vuole realizzare un dict persona() ed i dati inseriti all'interno non saranno fissi come visto negli esempi delle altre lezioni ma dovranno essere inseriti da utente. Dobbiamo avere l'opportunità di aggiungere, rimuovere e modificare delle eventuali chiavi. Incominciamo con il creare il dict persona

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}
```

Ora voglia creare una tupla dove inserire delle operazioni tipo aggiungere, rimuovere ecc… Viene creata appositamente una tupla in quanto non si può modificare

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

operazioni = ("aggiungere", "modificare", "eliminare")
```

Dato che dobbiamo gestire tutto da programma definiamo uno start che ci chieda che operazione vogliamo eseguire ed in base a cosa vogliamo fare ci dia pass (per aggiungere e modificare) oppure faccia delle altre operazioni per eliminare, anche se al momento teniamo anche qui il pass.

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

operazioni = ("aggiungere", "modificare", "eliminare")

def start():
    operazione = input("Cosa vuoi fare? ")
    if operazione == operazioni[0]:
        pass
    elif operazione == operazioni[1]:
        pass
    elif operazione == operazioni[2]:
        pass
```

Vogliamo che questo programma sia ciclico, nel senso che ci chieda fino all'infinito cosa vogliamo fare, motivo per cui mettiamo un while

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

operazioni = ("aggiungere", "modificare", "eliminare")

def start():
    operazione = input("Cosa vuoi fare? ")
    if operazione == operazioni[0]:
        pass
    elif operazione == operazioni[1]:
        pass
    elif operazione == operazioni[2]:
        pass

while True:
    start()
```

Per terminare il programma dobbiamo premere ctrl+c. In alternativa possiamo aggiungere ad operazioni() un nuovo comando che potremmo chiamare "termina" ed aggiungere a start un altro elif che quando vede la scritta termina stoppa il programma.

Torniamo in start ed invece di pass cominciamo a mettere le operazioni da eseguire. Cominciamo da "aggiungere".

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

operazioni = ("aggiungere", "modificare", "eliminare")

def start():
    operazione = input("Cosa vuoi fare? ")
    if operazione == operazioni[0]:
        x = input("Aggiungi chiave:valore separati da una virgola: ")
        aggiungi() # ancora, però non abbiamo definito aggiungi
    elif operazione == operazioni[1]:
        pass
    elif operazione == operazioni[2]:
        pass

while True:
    start()
```

Dobbiamo, ora, definire i parametri nel metodo aggiungi(), per cui definiamo proprio aggiungi() con all'interno i il parametro param che serve per dare l'input dell'utente. Se ad esempio vogliamo inserire telefono e numero di telefono possiamo inserire un solo parametro diviso dalla virgola (telefono,123456). Usando uno split sulla virgola, sappiamo che quello che c'è prima della virgola è la stringa telefono e quello che c'è dopo è il numero

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

operazioni = ("aggiungere", "modificare", "eliminare")

def start():
    operazione = input("Cosa vuoi fare? ")
    if operazione == operazioni[0]:
        x = input("Aggiungi chiave:valore separati da una virgola: ")
        aggiungi(x.split(","))
    elif operazione == operazioni[1]:
        pass
    elif operazione == operazioni[2]:
        pass

def aggiungi(param):
    pass


while True:
    start()
```

Andiamo a completare il metodo aggiungi sapendo che è formato da param[0] che è la chiave e cioè tutto quello che c'è prima della virgola, e param[1] che è il valore ed è tutto quello che c'è dopo la virgola

```
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

operazioni = ("aggiungere", "modificare", "eliminare")

def start():
    operazione = input("Cosa vuoi fare?: ")
    if operazione == operazioni[0]:
        x = input("Aggiungi chiave:valore separati da una virgola: ")
        aggiungi(x.split(","))
    elif operazione == operazioni[1]:
        pass
    elif operazione == operazioni[2]:
        pass

def aggiungi(param):
    chiave = param[0]
    valore = param[1]
    persona[chiave] = valore
    print(persona)

while True:
    start()
```

Testiamo il programma fino a qua mandandolo in esecuzione:

```
Cosa vuoi fare?: aggiungere
Aggiungi chiave:valore separati da una virgola: telefono,123456
{'nome': 'Luca', 'cognome': 'Rossi', 'eta': 25, 'telefono': '123456'}
Cosa vuoi fare?:
```

Dunque alla prima domanda abbiamo risposto *aggiungere* che ci ha permesso di andare alla seconda domanda. Alla seconda domanda abbiamo dato il parametro *telefono,123456* che grazie alla virgola è stato splittato in 2 valori, chiave:valore appunto. Il programma ha così aggiunto il telefono alla persona esistente Luca Rossi 25. Da notare che il programma richiederà di nuovo cosa vogliamo fare, lo farà all'infinito fino a quando non premiamo ctrl+c oppure definiamo un metodo termina che appena scriviamo termina si ferma. L'esercizio dovrà essere completato con i metodi "modificare" ed "eliminare".

```python
persona = {
    "nome": "Luca",
    "cognome": "Rossi",
    "eta": 25
}

operazioni = ("aggiungere", "modificare", "eliminare")

def start():
    operazione = input("Cosa vuoi fare?: ")
    if operazione == operazioni[0]:
        x = input("Aggiungi chiave:valore separati da una virgola: ")
        aggiungi(x.split(","))
    elif operazione == operazioni[1]:
        chiave = input("Modifica chiave: ")
        nuovo_valore = input("Nuovo valore: ")
        modificare(chiave, nuovo_valore)
    elif operazione == operazioni[2]:
        chiave = input("Elimina chiave: ")
        eliminare(chiave)

def aggiungi(param):
    chiave = param[0]
    valore = param[1]
    persona[chiave] = valore
    print(persona)

def modificare(chiave, nuovo_valore):
    if chiave in persona:
        persona[chiave] = nuovo_valore
        print(persona)
    else:
        print(f"La chiave '{chiave}' non esiste nella persona.")

def eliminare(chiave):
    if chiave in persona:
        del persona[chiave]
        print(persona)
    else:
        print(f"La chiave '{chiave}' non esiste nella persona.")

while True:
    start()
```

In questa ultima soluzione, la completa, sono presenti delle parentesi graffe che racchiudono “chiave” all’interno del print. Il perché lo vediamo nella seguente lezione con lo string formatting.

LEZIONE 28

FORMATTAZIONE STRINGA AVANZATA:

Delle stringhe è già stato parlato abbondantemente nelle lezioni precedenti. In questa lezione viene esplicitata la formattazione delle stringhe. Il "formatting" consiste nello strutturare delle stringhe in un certo modo partendo da un altro formato. Facciamo un esempio:

```
peso = 75
altezza = 176

frase = "Ciao sono Marco e sono alto " + str(altezza) + " cm"

print(frase)
```

```
Ciao sono Marco e sono alto 176 cm
```

Di sicuro il risultato è corretto ma trasformare un intero (altezza) in stringa per poterlo concatenare in frase che è formata da stringa per poi stamparlo non è il massimo per un programmatore. Ecco che ci viene in aiuto lo string formatting:

```
peso = 75
altezza = 176

frase = "Ciao sono Marco e sono alto {} cm"

print(frase.format(altezza))
```

```
Ciao sono Marco e sono alto 176 cm
```

In pratica mettendo delle parentesi graffe all'interno della stringa ci permette di inserire dei valori che possono essere richiamati successivamente.

All'interno delle parentesi potremmo utilizzare anche dei parametri, tipo **:.2f** che significa metti 2 zeri dopo la virgola.

```
peso = 75
altezza = 176

frase = "Ciao sono Marco e sono alto {:.2f} cm"

print(frase.format(altezza))
```

```
Ciao sono Marco e sono alto 176.00 cm
```

Se si vogliono conoscere tutte le **string format parameters in Python** basta fare una ricerca e verranno visualizzati facilmente. In questo caso i **:** indicano che c'è un parametro; il **.** indica che c'è una virgola; **2f** indica due zeri dopo la virgola.

Vediamo ora come **inserire valori multipli**, ovvero più parametri all'interno della stessa frase.

```
peso = 75
altezza = 176
anni = 25

frase = "Ciao sono Marco, ho {} anni, sono alto {} cm e peso {} Kg"

print(frase.format(anni, altezza, peso))
```

```
Ciao sono Marco, ho 25 anni, sono alto 176 cm e peso 75 Kg
```

Notiamo che in print dobbiamo inserire i parametri in ordine altrimenti si confonderebbero peso, altezza ed età. Ci vengono in soccorso gli indici. Se indicizziamo i 3 argomenti con 0-1-2, dove lo 0 corrisponde al primo parametro messo in print, l'1 al secondo e così via, possiamo spostare gli argomenti all'interno della frase senza alcun problema tanto con gli indici sappiamo sempre il valore giusto da inserire:

```
peso = 75
altezza = 176
anni = 25

frase1 = "Ciao sono Marco, ho {0} anni, sono alto {1} cm e peso {2} Kg"
frase2 = "Ciao sono Marco, sono alto {1} cm, ho {0} anni,  e peso {2} Kg"

print(frase1.format(anni, altezza, peso))
print(frase2.format(anni, altezza, peso))

# nei print sia in frase1 che in frase2 sono stati messi volutamente
# gli argomenti con lo stesso ordine (anni, altezza, peso)
# per far notare che cambiando l'ordine della frase2 rispetto alla frase1
# ma mantenedo gli indici, gli argomenti vengono comunque
# posizionati nel'ordine corretto
```

```
Ciao sono Marco, ho 25 anni, sono alto 176 cm e peso 75 Kg
Ciao sono Marco, sono alto 176 cm, ho 25 anni,  e peso 75 Kg
```

Altri indici che ci vengono in aiuto sono gli **indici nominali**.

```
peso = 75
altezza = 188
anni = 25

frase = "Ciao sono Marco, ho {anni} anni, sono alto {altezza} cm e peso {peso} Kg"

print(frase.format(anni=anni, altezza=altezza, peso=peso))
```

```
Ciao sono Marco, ho 25 anni, sono alto 188 cm e peso 75 Kg
```

LEZIONE 29

LAVORARE CON I FILE:

Incominciamo a vedere come gestire i file (handling). Cominciamo a vedere come **aprire un file attraverso 4 parametri (r, a, w, x)**:

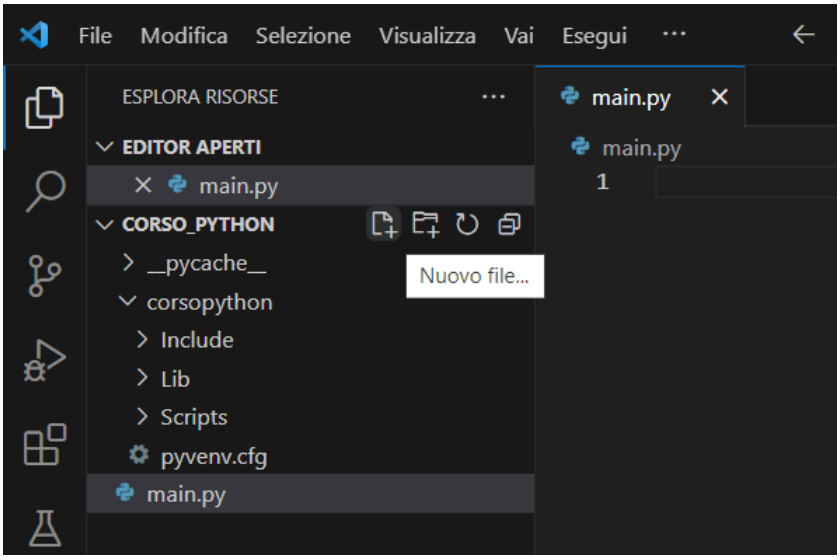
r – Read: apre il file per leggerlo; restituisce un errore se il file non esiste

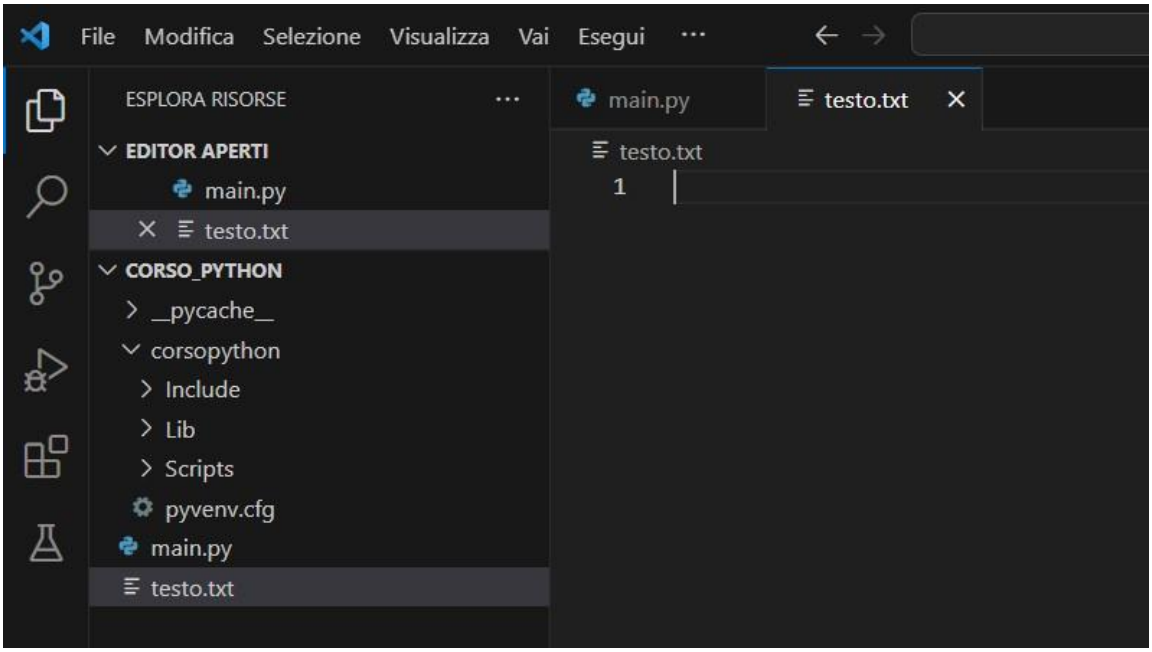
a – Append: apre il file per appendere in fondo (quindi solo accodare nuovo codice); crea il file se non esiste

w – Write: apre il file per scrivere in qualunque parte del file; crea il file se non esiste

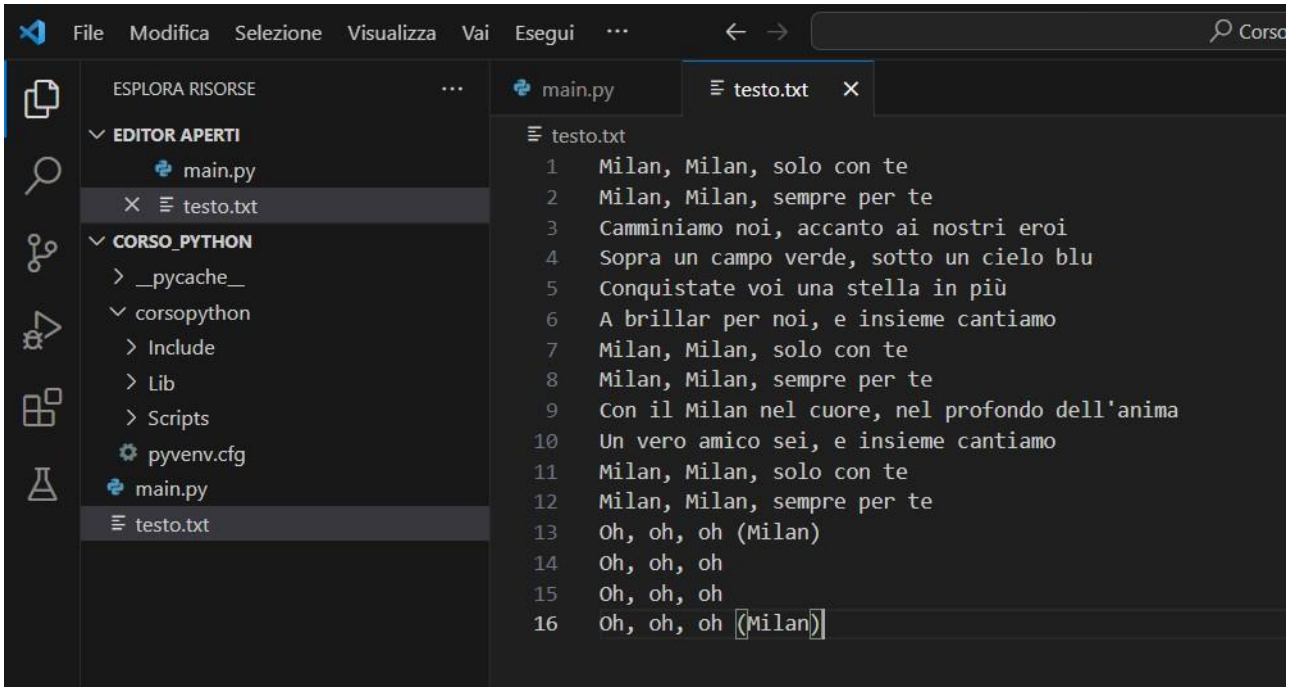
x – Create: crea il file; restituisce un errore se il file non esiste

Cominciamo con il creare un file. Andiamo a sinistra e sotto Corso_Python, selezioniamo il Main e clicchiamo su "nuovo file" e lo chiamiamo **testo.txt**





All’interno del file scriviamo qualche frase per poi poterci lavorare:



Ora andiamo nel **Main** e lo apriamo in modalità lettura (r):

```
f = open("testo.txt")
print(f.read())
```

```
Milan, Milan, solo con te
Milan, Milan, sempre per te
Camminiamo noi, accanto ai nostri eroi
Sopra un campo verde, sotto un cielo blu
Conquistate voi una stella in piu'
A brillar per noi, e insieme cantiamo
Milan, Milan, solo con te
Milan, Milan, sempre per te
Con il Milan nel cuore, nel profondo dell'anima
Un vero amico sei, e insieme cantiamo
Milan, Milan, solo con te
Milan, Milan, sempre per te
Oh, oh, oh (Milan)
Oh, oh, oh
Oh, oh, oh
Oh, oh, oh (Milan)
```

Per **leggere una parte di file** e non tutto il testo basta specificare il numero di caratteri da leggere:

```
f = open("testo.txt")
print(f.read(5))
```

```
Milan
```

Se si ha un **testo con più righe, per leggere la prima riga**:

```
f = open("testo.txt")
print(f.readline())
```

```
Milan, Milan, solo con te
```

Se facciamo più print readline abbiamo a schermo non solo la prima riga ma in maniera sequenziale anche le successive:

```
f = open("testo.txt")

print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())
```

```
Milan, Milan, solo con te

Milan, Milan, sempre per te

Camminiamo noi, accanto ai nostri eroi

Sopra un campo verde, sotto un cielo blu

Conquistate voi una stella in piu'
```

Ha più senso, ovviamente, se volessimo passare riga per riga scrivere un for piuttosto che tanti print:

```
f = open("testo.txt")

for riga in f:
    print(riga)
```

```
Milan, Milan, solo con te

Milan, Milan, sempre per te

Camminiamo noi, accanto ai nostri eroi

Sopra un campo verde, sotto un cielo blu

Conquistate voi una stella in piu'

A brillar per noi, e insieme cantiamo

Milan, Milan, solo con te

Milan, Milan, sempre per te

Con il Milan nel cuore, nel profondo dell'anima

Un vero amico sei, e insieme cantiamo

Milan, Milan, solo con te

Milan, Milan, sempre per te

Oh, oh, oh (Milan)

Oh, oh, oh

Oh, oh, oh

Oh, oh, oh (Milan)
```

Una volta letto il file conviene sempre chiuderlo. Da tenere presente che una volta chiuso se si va a leggere avremo un errore. Per leggerlo di nuovo va riaperto.

```
f = open("testo.txt")

for riga in f:
    print(riga)

f.close()

print(f.readline())
```

```
ValueError: I/O operation on closed file.
```

Vediamo come **scrivere in un file**. Intanto, per semplificarci le cose, per non avere troppo testo lasciamo solo 2 righe nel file. In **testo.txt** avremo:

```
Milan, Milan, sempre per te
Camminiamo noi, accanto ai nostri eroi
```

Nel **main** utilizziamo la a di append per scrivere:

```
f = open("testo.txt", "a") # la r di read si può omettere, le altre lettere no
f.write(" SCRIVIAMO COSE A CASO")
f.close
```

Torniamo in **testo.txt** ed avremo:

```
Milan, Milan, sempre per te
Camminiamo noi, accanto ai nostri eroi SCRIVIAMO COSE A CASO
```

Dato che il file è chiuso, se vogliamo leggerlo nel main basta riaprirlo e leggerlo:

```
f = open("testo.txt", "a") # la r di read si può omettere, le altre lettere no
f.write(" SCRIVIAMO COSE A CASO")
f.close

f = open("testo.txt", "r")
print(f.read())
f.close()
```

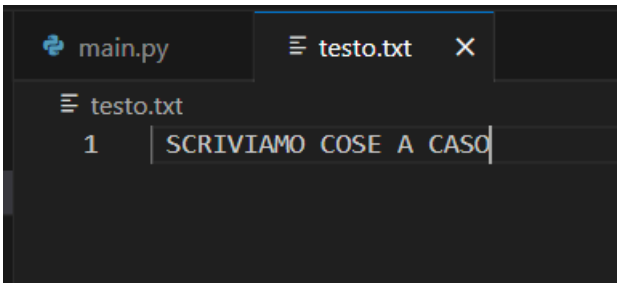
```
Milan, Milan, sempre per te
Camminiamo noi, accanto ai nostri eroi SCRIVIAMO COSE A CASO
```

Se vogliamo **sovrascrivere** tutto il file allora usiamo w (write):

```
f = open("testo.txt", "w")
f.write(" SCRIVIAMO COSE A CASO")
f.close

f = open("testo.txt", "r")
print(f.read())
f.close()
```

```
 SCRIVIAMO COSE A CASO
```

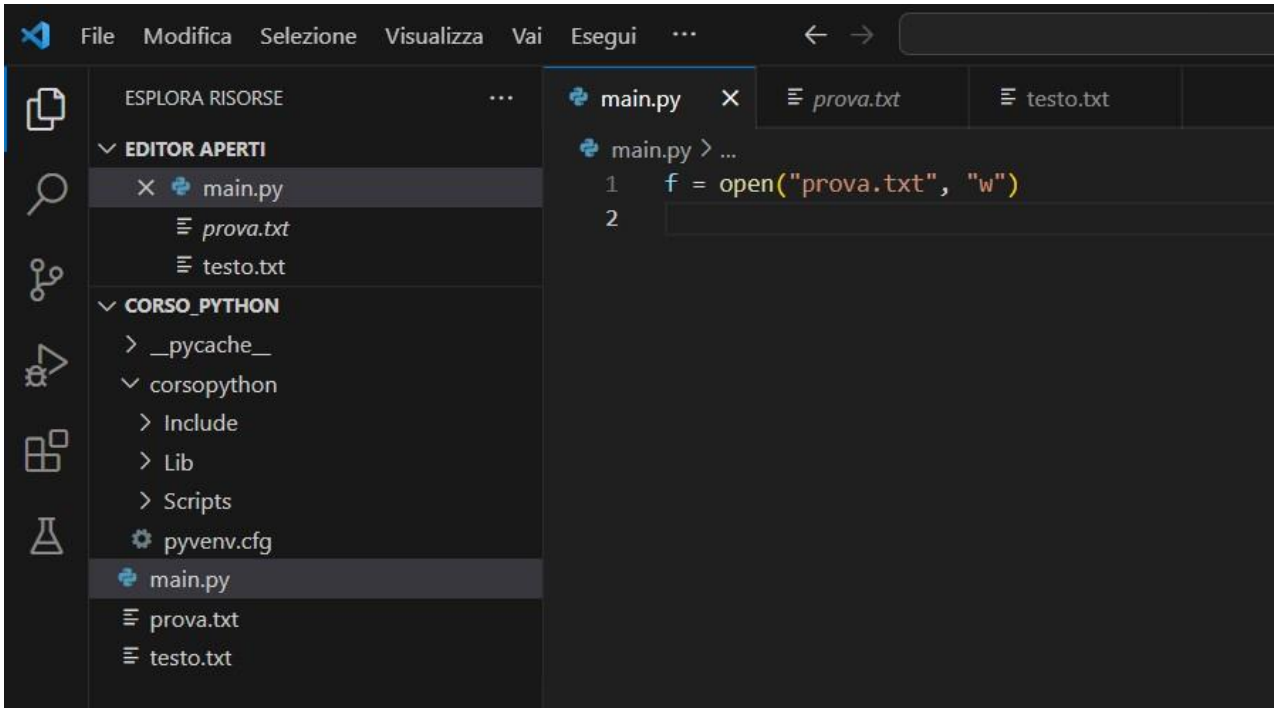
Ora nel file di testo avremo questa scritta e non più le 2 righe di prima.



Vediamo ora come creare un file. Si può usare x, a, w. Creiamo il file prova.txt ed utilizziamo w:

```
f = open("prova.txt", "w")
```

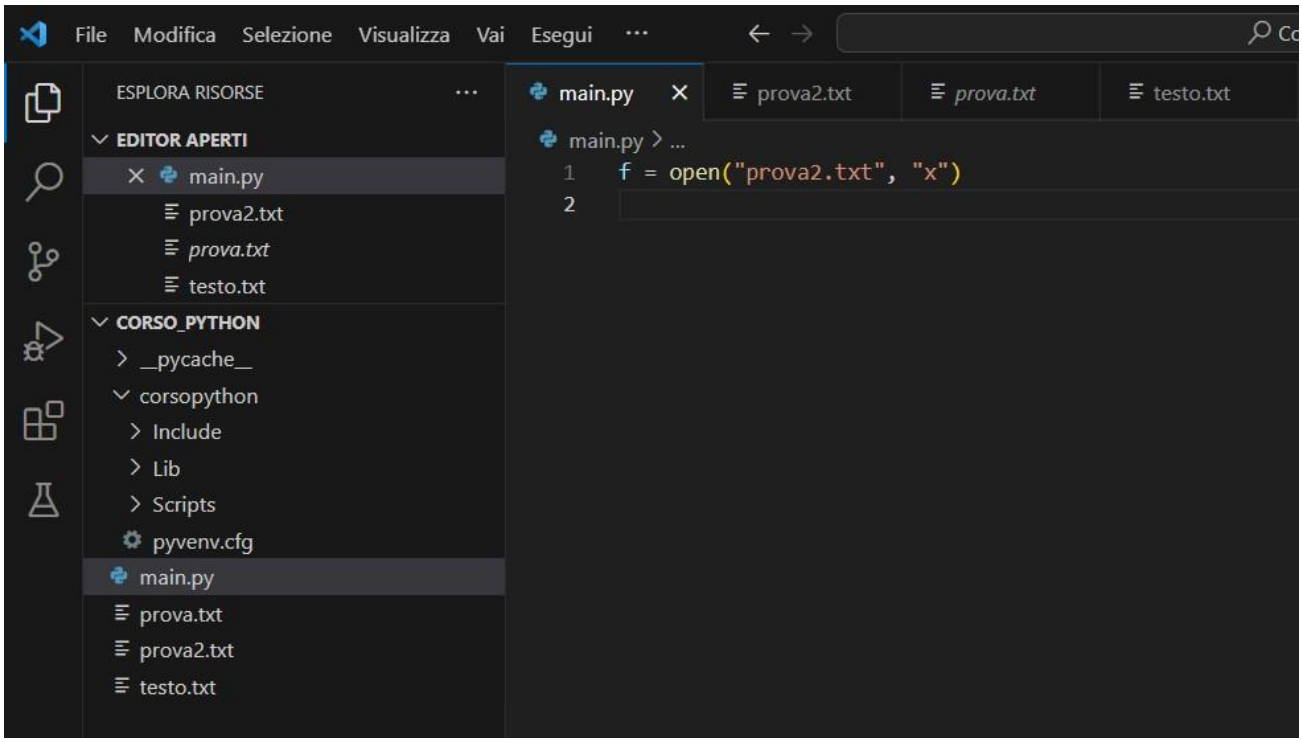
otteniamo un nuovo file vuoto:



La stessa cosa vale se vogliamo creare un file con x:

```
f = open("prova2.txt", "x")
```

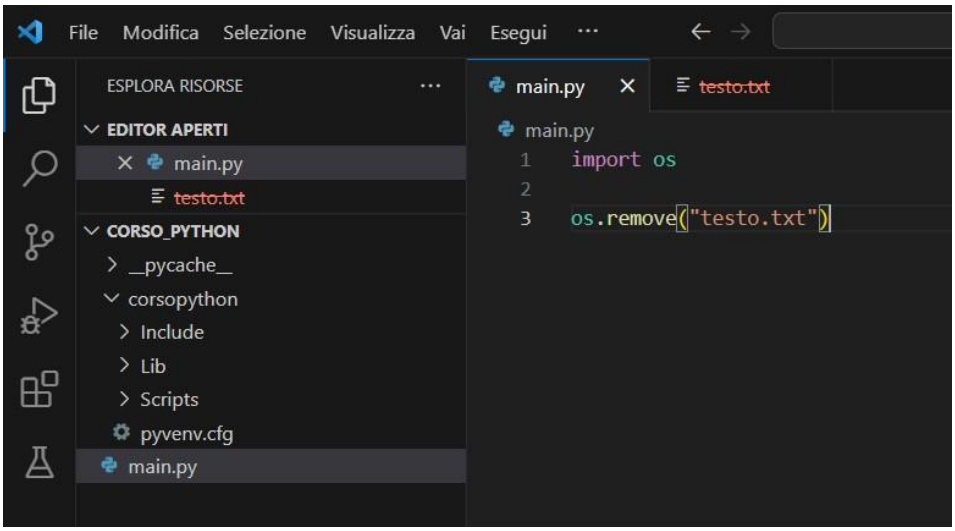
ottenendo sempre un file vuoto:



Vediamo ora come **eliminare un file**. Per gestire l'eliminazione bisogna importare il modulo os:

```
import os

os.remove("testo.txt")
```

e vediamo che il file scompare:



Per **verificare se un file esiste ed eventualmente rimuoverlo** possiamo usare un if:

```
import os

if os.path.exists("prova.txt"):
    os.remove("prova.txt")
else:
    print("file inesistente")
```

```
file inesistente
```

Con lo stesso principio si possono creare o eliminare le cartelle invece che i file. A titolo di esempio per rimuovere una ipotetica cartella "lavoro" dobbiamo importare os ed usare il comando rmdir (remove directory)

```
import os

os.rmdir("lavoro")
```

LEZIONE 30

SITO DI RIFERIMENTO PER MOLTI MODULI:

Per approfondimenti sui vari moduli (tipo NumPy per gli array) si può consultare il sito:

https://www.w3schools.com/python/default.asp